Mining and Untangling Change Genealogies

Kim Sebastian Herzig

Dissertation zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften
der Naturwissenschaftlich-Technischen Fakultäten
der Universität des Saarlandes
Saarbrücken, 2012

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet. Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form in einem Verfahren zur Erlangung eines akademischen Grades vorgelegt.

Saarbrücken, den 6.5.2013

Kim Herzig

Abstract

Developers change source code to add new functionality, fix bugs, or refactor their code. Many of these changes have immediate impact on quality or stability. However, some impact of changes may become evident only in the long term. This thesis makes use of *change genealogy* dependency graphs modeling dependencies between code changes capturing how earlier changes enable and cause later ones. Using change genealogies, it is possible to:

(a) apply formal methods like *model checking* on version archives to reveal temporal process patterns. Such patterns encode key features of the software process and can be validated automatically: In an evaluation of four open source histories, our prototype would recommend pending activities with a precision of 60—72%.

(b) *classify the purpose of code changes*. Analyzing the change dependencies on change genealogies shows that change genealogy network metrics can be used to automatically separate bug fixing from feature implementing code changes.

(c) *build competitive defect prediction models*. Defect prediction models based on change genealogy network metrics show competitive prediction accuracy when compared to state-of-the-art defect prediction models.

As many other approaches mining version archives, change genealogies and their applications rely on two basic assumptions: code changes are considered to be atomic and bug reports are considered to refer to corrective maintenance tasks. In a manual examination of more than 7,000 issue reports and code changes from bug databases and version control systems of open- source projects, we found 34% of all *issue reports to be misclassified* and that up to 15% of all applied *issue fixes consist of multiple combined code changes* serving multiple developer maintenance tasks. This introduces bias in bug prediction models confusing bugs and features. To partially solve these issues we present an approach to untangle such combined changes with a mean success rate of 58—90% after the fact.

Zusammenfassung

Softwareentwickler ändern Source-Code um neue Funktionalität hinzuzufügen, Bugs zu beheben oder um ihren Code zu restrukturieren. Viele dieser Änderungen haben einen direkten Einfluss auf Qualität und Stabilität des Softwareprodukts. Jedoch kommen einige dieser Einflüsse erst zu einem späteren Zeitpunkt zur Geltung. Diese Arbeit verwendet *Genealogien* zwischen Code-Änderungen um zu erfassen, wie frühere Änderungen spätere Änderungen erfordern oder ermöglichen. Die Verwendung von Änderungs-Genealogien ermöglicht:

(a) die Anwendung formaler Methoden wie *Model-Checking* auf Versionsarchive um temporäre Prozessmuster zu erkennen. Solche Prozessmuster verdeutlichen Hauptmerkmale eines Softwareentwicklungsprozesses: In einer Evaluation auf vier Open-Source Projekten war unser Prototyp im Stande noch ausstehende Änderungen mit einer Präzision von 60–72% vorherzusagen.

(b) *die Absicht einer Code-Änderung zu bestimmen.* Analysen von Änderungsabhängigkeiten zeigen, dass Netzwerkmetriken auf Änderungsgenealogien geeignet sind um fehlerbehebende Änderungen von funktionalitätshinzufügenden Änderungen zu trennen.

(c) *konkurrenzfähige Fehlervorhersagen zu erstellen.* Fehlervorhersagen basierend auf Genealogie-Metriken können sich mit anerkannten Fehlervorhersagemodellen messen.

Änderungs-Genealogien und deren Anwendungen basieren, wie andere Data-Mining Ansätze auch, auf zwei fundamentalen Annahmen: Code-Änderungen beabsichtigen die Lösung nur eines Problems und Bug-Reports weisen auf Fehler korrigierende Tätigkeiten hin. Eine manuelle Inspektion von mehr als 7.000 Issue-Reports und Code-Änderungen hat ergeben, dass 34% aller Issue-Reports falsch klassifiziert sind und dass bis zu 15% aller fehlerbehebender Änderungen mehr als nur einem Entwicklungs-Task dienen. Dies wirkt sich negativ auf Vorhersagemodelle aus, die nicht mehr klar zwischen Bug-Fixes und anderen Änderungen unterscheiden können. Als Lösungsansatz stellen wir einen Algorithmus vor, der solche nicht eindeutigen Änderungen mit einer Erfolgsrate von 58–90% entwirrt.

Contents

Chapter 1

Introduction

Software development is an incremental process in which developers apply changes to the current version of a software project to achieve the next advancement towards a milestone or release. Each change is supposed to add new value to the software: adding a new feature, fixing a bug, adapting the software to new environments, or simply to clean up the code base. Some of the changes made by the developers introduce new issues and defects. Fixing those defects gets increasingly expensive as the software development process progresses [97]. Thus, one goal of quality assurance is to detect as many defects as possible and fix them in the earlier stages of development. To increase quality assurance effectiveness, it is crucial to focus on those components that are most likely to contain defects.

Software and its reliability is a product of its history. Therefore it is important to analyze and understand the project history. The goal is to gather evidence on good and bad practices that influence the quality of the corresponding software system—evidence on the effectiveness of testing, the quality of bug reports, the role of complexity metrics, the influence of code changes on each other, and so on. Software archives, such as version control systems and issue tracking systems, are a rich source of such evidence. These archives record many of the activities around a software product including problem descriptions and code changes applied. This evidence can be used to build powerful and effective tools to support developers helping them to write better source code inducing fewer defects. Supporting quality assurance, the evidence can also be used to detect and predict defect prone code entities that should be target of careful and excessive quality testing.

This thesis makes two major contributions to mining version archives. First and foremost, we introduce the concept of change dependency graphs called *change genealogies* that combine two popular but limited visions on version control systems. Many research studies in analyzing software history have been mostly constrained to either space or time. Being constrained to space means that one examines the evolution of single components, aggregating features over time. Being constrained to time means that one examines which components were changed at a single moment in time, extracting co-changes from the resulting change sets. What we would like to have is reasoning over multiple components at multiple points in time. Change genealogies allow such multi-dimensional reasoning but also allow the use of formal methods such as model checking and the use of temporal logic formulas like CTL to express both temporal and spatial patterns.

The second contribution of this thesis targets the threats of noise and bias in common mining data sets. The tremendous progress gained in the research field of empirical software engineering and in particular in the mining software repository community is raising ever-growing concerns about the amount of noise in data sets and the impact of such noise on any mining model based on such data sets. During our experiments we identified two major noise factors in common mining data sets: *tangled code changes* and *misclassified issue reports*. This thesis discusses these two common noise factors, shows their impact on quality models, and provides approaches to reduce the amount of noise and bias within these common mining data sets.

More detailed, this work makes the following contributions to the research field of mining version archives.

Change genealogies We introduce the concept of *change genealogies* that model dependencies between code changes adding, modifying, or deleting methods and method calls as graph structure.

Evaluation of bug data In a large scale manual inspection of over 7,000 bug reports, we collected evidence that bug databases contain large fractions of misclassified bug reports threatening state-of-the-art mining models based on such noised and biased data sets. The manual classified data set is public available and can be used as ground truth for further research.

Tangled code changes Change genealogies and other mining models assume code changes to be atomic—code changes serving different developer maintenance tasks are applied separately. This thesis provides evidence that there exist a large fraction of non-atomic code changes that threaten state-of-the-art defect prediction models and mining models.

Untangling code changes To address this issue of tangled code changes, we introduce an algorithm that will partition tangled code changes into individual, non-overlapping sub-changes each suspected to address a separate developer maintenance task.

Change genealogy metrics Based on the change genealogy graphs, we compute network metrics that can be used to distinct between bug fixing and feature adding code changes and that allow the construction of precise defect prediction models.

Long-term cause effect chains By mining and model checking change genealogies, we obtain frequent temporal patterns that encode key features of the software process that span both space and time.

1.1 Thesis Structure

The main contributions of this thesis are the introduction of change genealogies to model change dependencies, to improve state-of-the-art mining approaches adding additional change dependency information, and to partially solve issues regarding noise and bias in software repository data sets and their impact on common mining approaches and techniques. The remainder of this thesis is structured as follows:

Chapter 2 gives a short introduction on mining software archives, related definitions and related work. After discussing concepts and giving a short overview over common mining techniques, we will focus on three common assumptions and hypotheses many mining approaches rely on and that threaten any mining approach if invalid.

Chapter 3 introduces the concept of change genealogies modeling temporal and structural dependencies between code changes or sets of code changes.

Chapter 4 will then follow up on the noise and bias discussion to investigate the reliability of issue report types mined from issue tracking systems. Our investigation shows that there exist a critical amount of falsely classified bug reports threatening any mining model not performing extra bug data validation. The public available data set containing our manual inspection results can be used as ground truth for further research.

Chapter 5 focuses on the issue of tangled code changes—changes being a mixture of several developer developer maintenance tasks. The study provides evidence of the existence and spreading of tangled changes and their potential impact on common mining models. We also provide an *untangling algorithm* that partitions tangled sets of code changes into smaller, non-overlapping partitions. Each code change partition contains a set of code changes that are likely to depend on each other and thus are likely to be necessary to resolve the same developer maintenance task. Our approach untangles any two artificially tangled change sets with a precision between 0.67 and 0.93 and reduces the number of source files falsely associated to developer maintenance tasks by 55% to 81%.

Chapter 6 describes a set of change genealogy network metrics used to automatically detect bug fixing code changes and to predict defective source files. Bug fixing code changes show significant different structural dependencies to other code changes when compared to feature adding code changes. Change genealogy metrics can be used to express these dependency differences to build appropriate classifiers. Further, we show that we can use change genealogy metrics to predict defective source files.

Chapter 7 shows how to use change genealogies to apply model checking to version archives. Using the expressive power of CTL formulas and a directed acyclic change genealogy we were able to predict long-term cause effect chains between code changes. Each reported long-term coupling implies a structural dependency between coupled artifacts. Using additional change properties as coupling conditions we were able to detect change coupling CTL rules that cannot be detected by comparable approaches.

Chapter 8 concludes with a summary of our results, lessons learned, and ideas for future work.

1.2 Publications

This dissertation builds on the following papers (in chronological order):

- **Kim Herzig**.
  Capturing the long-term impact of changes. In Proceedings of the *32nd International Conference on Software Engineering* – Volume 2 pages 393–396 New York, NY, USA, 2010. ACM.

- **Kim Herzig** and Andreas Zeller.
  Mining Cause-Effect-Chains from Version Histories. In Proceedings of the *22nd International Symposium on Software Reliability Engineering* pages 60–69 Washington, DC, USA, 2011. IEEE Computer Society.

- **Kim Herzig**, Sascha Just, and Andreas Zeller.
  It's not a Bug, it's a Feature: How Misclassification Impacts Bug Prediction. In Proceedings of the *35nd International Conference on Software Engineering* New York, NY, USA, 2013. ACM.

The following technical reports are public available and under submission:

- **Kim Herzig** and Andreas Zeller.
  Untangling Changes. In Proceedings of the *10th Working Conference on Mining Software Repositories* San Francisco, California, USA, 2013. ACM.

- **Kim Herzig**, Sascha Just, Andreas Rau, and Andreas Zeller.
  Classifying Changes and Predicting Defects using Change Genealogies. Submitted to *9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE '13).*

All experiments and experimental setups presented and discussed in this thesis are designed and conducted by the author of this thesis.

Chapter 2

Background

2.1 Software Repositories

Developing and using software can produce large amounts of artifacts over years of project history: problems occurred, changes applied, problems fixed, documentation updated, etc. Most of these artifacts get collected and stored, at least temporarily. Storages for such artifacts are called *software repositories*—an umbrella term for repositories containing different kinds of artifacts produced during developing and usage of the software. One of the most prominent software repository types is the version control system (e.g. `Git` [1], `Mercurial` [3], `Subversion` [4]) that contains all changes applied to the code base of the software project.

Fortunately, software repositories are also a relative inexpensive way to gather lots of information about the software project and its development process. Using these important sources of historic data it is possible to recap most of the project history and to extract frequently occurring problems and patterns. Software repositories provide a good basis to learn from history without setting up additional infrastructure, disturbing developers, or to slow down development. In this work, we will concentrate mostly on two software repositories: version control systems and issue tracking systems.

2.1.1 Version Control Systems

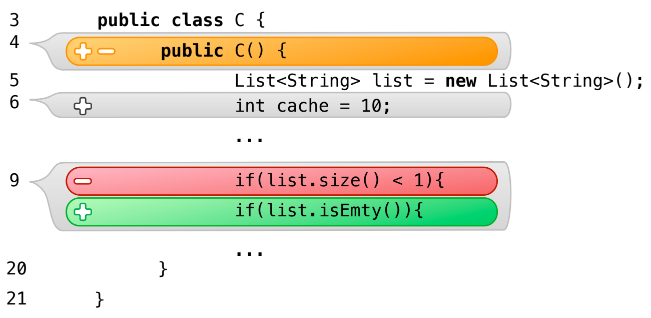
Version control systems record changes applied to documents and files. In software projects, most of these changes are applied to source files modifying the code base of the software system and therefore also change the runtime behavior of the software product. Most version control systems also store meta data attached to each file changed. Prominent examples are the author (username, email or full name), the timestamp of the change, and a short message—usually called commit message—given by the author of the change. Although different version control systems are based on different concepts, most of these systems group simultaneously applied changes into *change sets*. Thus, a change set contains multiple code changes to at least one file that got committed simultaneously. Author, timestamp, and commit message are annotating change sets. Each change set is identified using a unique identification number or string. Using this identifier, one can restore the files to the state when this revision reflected the most current version of the software.

2.1.2 Change Sets

In practice, changes to source code are not applied separately but would be committed together. Ideally, code changes belonging to the same logical unit or development task are committed together. Committing a set of code changes creates a snapshot of the software project that can be restored any time later. Depending on the version control system implementation, committing code changes might make the applied code changes public available and might require other developers to update their code base according to your applied patch. Version control systems like `Git` [1] require an explicit command in order to publish committed code changes. For the remainder of this thesis, we use the term *change sets* for the set of simultaneously committed code changes.

2.1.3 Change Operations

Version control systems record changes applied to any file. In the context of this work, version control systems record changes to files containing source code. Changes to files not containing source code will be ignored, unless stated otherwise. Thus, analyzing version control systems requires analyzing code changes on a predefined level of



```
public class C {
    public C() {
            List<String> list = new List<String>();
            int cache = 10;
            ...
            if(list.size() < 1){
            if(list.isEmty()){
            ...
    }
}
```

Figure 2.1: Sample diff output containing three change operations and two add/delete change operations.

granularity. Many `diff` tools (e.g. *GNU Diffutils*[1]) output differences line by line. All changes applied within the same line are combined into a single change replacing the old line by a new line. Other tools like LSDiff [85, 49] work on structural differences on source code files to extract more fine-grained source code changes.

In this thesis, we define *change operation* as a set of source code changes that added or deleted method calls and definitions, or that modified method bodies. Method definitions are identified using their full-qualified name[2]. Method calls are identified using their absolute position in the source code[3]. We call a set of source code changes that added or deleted method calls or method definitions *add/delete change operation*. Add/delete change operations do not include modifications of method definitions and bodies. Thus each change set corresponds to a set of change operations adding or deleting a method definition (AD, DD), modifying a method body (MD), or a method call (AC, DC). Analog, each add/delete change operation corresponds to a set of change operations adding or deleting a method definition (AD, DD) or a method call (AC, DC). Thus, an add/delete change operation is a subset of the corresponding change operation.

The change set shown in Figure 2.1 derives a set containing three change operations. One DC operation that deletes the method call `list.size()`, one AC operation adding `list.isEmpty()`, and one MD operation modifying the constructor `public C(){...}`. The MD operation is contained in the set of change operations, only. The corresponding add/delete change operations contain only the DC and DC operations.

[1]http://www.gnu.org/software/diffutils/
[2]in Java: package name + class name + method name + method signature.
[3]full-qualified source file name + character position in source file

Further, changing a method body by adding a method call will result in two change operations, one MD and one AC. Renaming of method definition or moving a defined method to a super or sub class is represented using a method definition deletion (DD) and a method definition addition (AD).

2.1.4 Issue tracking systems

Issue tracking systems are used to keep track of developer tasks and program issues. In most projects, the majority of recorded issues are so called *bug reports*—software problems and defects found by users and developers. But there can be many other issue types such as tasks, feature requests, documentation issues, questions, and specification issues. Each issue can be assigned to individual members of the development team, contains multiple fields that express specific environmental setups, but also provide the possibility using comments attached to each issue.

2.2 Software Metrics

Source code is structured text that can be interpreted by a compiler or an interpreter. But how do you argue about the quality of this structured text, its maintainability, or its complexity. We need objective, reproducible, and quantifiable measurements that allow us to argue about specific properties of source code. These measurements are typically called *software metrics*. One of the most common software metrics measured the length of source code (e.g. lines of code or lines of statements). Software metrics are used in many applications and allow a numeric representation of source code properties. Hundreds of software metrics are used every day. The studies and approaches presented in this thesis are using many different software metrics that are suitable to express the complexity, maintainability, stability, or quality of a source code artifact. The number of defects fixed in a source code artifact can be seen as a software metric.

The classical software metric is based on the source code or an individual source code artifact. But frequently, these classical software metrics are extended or even replaced by software metrics based on the development process and dependencies between artifacts. While development process metrics focus on the history of the source code (see Section 2.2.2) most dependency metrics are based on network graph structures [126, 151, 22] that express interactions between source code artifacts (e.g. method calls or `include` statements). These networks are similar to network models extracted known from social media applications and allow the use of communication and social media metrics.

It is important to realize that the set of software metrics that can used to successfully build recommendation or prediction models is highly dependent on the software projects, the analyzed programming language, and the purpose the software metrics will be used for. For instance, building defect prediction models for different projects usually requires different sets and combinations of software metrics to optimize the performance of the resulting model. Studies building defect prediction models for different projects or even projects releases [40, 110, 148, 134] and studies on cross project defect prediction [119, 152] show that there is no single set of metrics that performs equally well on all projects and on all levels of granularity. This implies that different projects require software metrics. Source code changes over time and so does the performance of software metrics. Once an optimal set of software metrics is found one should check their performance regularly to ensure the accuracy of the models based on this feature set.

2.2.1 Complexity Metrics

One of the most prominent sets of software metrics are classical complexity metrics. The rational behind complexity metrics is that the complexity of source code can be expressed by the structure of the source code. The more complex the structure of the source code, the harder the logic behind it and the easier it is to make mistakes. Cyclomatic complexity [96] is one of the most prominent complexity metric. Roughly, cyclomatic complexity is computed on the control flow graph of the program counting the number of decision points (e.g. loops and if-statements). Complexity metrics are usually easy to understand, easy to implement, and easy to compute.

Although complexity metrics tend to be simple and very limited they have been successfully used in many application domains. Complexity metrics are frequently used to estimate code quality and to train defect prediction models [64, 74, 98, 108, 115, 153]. But complexity metrics have shown to be useful to measure program maintenance [79, 118, 122] and to estimate development efforts [6, 125, 131]. A survey on MICROSOFT developers conducted by Zimmermann et al. [148] (including the author of this thesis) showed that 91% of the interviewed developers consider complex and highly coupled code to be more defect prone.

2.2.2 Change History Metrics

Code complexity has shown to be important to explain and estimate the number of defects in a program. But code complexity is only one variable of the equation. Graves et al. [58] showed that metrics describing the history of the existing code base could also be used to train successful defect prediction models. *Change history metrics* such as the number of past changes applied to the source code are strong indicators for defects. Changing source code frequently increases the likelihood of defects to be introduced. Similar, Hassan et al. [65] and Kim et al. [89] developed defect prediction models based on the number of fixes applied to the source code. The more fixes were applied the higher the chance to detect more bugs in future. Such defect prediction models can be highly accurate. In fact, Moser et al. showed that "change data are effectively better indicators for the presence or absence of software defects than static code attributes" [102]. Lately, Nagappan et al. [110] (including the author of this thesis) introduced the concept of change bursts—many frequent changes to a source code artifact in short time. Change bursts indicate incomplete requirements, hairy bugs, or insufficient quality assurance. Again, change bursts make no use of code structure.

2.2.3 Other Code Metrics

There exist many other software metrics that are frequently used in mining models. In particular, the set of network metrics is important for this thesis.

Network metrics are software metrics that are computed on network graphs modeling dependencies between code artifacts such as graphs modeling method calls between classes or import structures between files. Schröter et al. [126] used import dependencies to predict failure prone entities at design time. Studies from Microsoft Research [105, 150] used code dependencies to successfully identify failure-prone binaries. Shin et al. [128] provided evidence that defect prediction models can benefit when adding calling structure metrics. Program dependence graphs [44, 62, 117] have been used to improve testing [13, 117, 123], debugging and maintenance [117, 123].

Network metrics are very popular in social science using graphs modeling the interactions between humans. Zimmermann and Nagappan [151] were among the first that transferred this concept of social metrics on code databases. The assumption is that code artifacts interact with each other (like human actors) by calling each others methods or accessing their variables. The resulting graph structure can be used to apply social network analysis on code dependency graphs and to reuse social network metrics on such code dependency graphs. The results presented by Zimmermann and Nagappan [151] show that network metrics can be very helpful when predicting defects and likewise. Later, Bird et al. [22] extended the set of network metrics by extending code dependency graph adding contribution dependency edges. Contribution edges model the dependency between code artifacts derived from the contribution history of a software project as described by Pinzger et al. [116]. Premraj and Herzig [119] showed that network metrics in a realistic defect prediction setup add no significant value compared to an experimental setup using complexity metrics, only. Still, network metrics overcome a drawback most software metrics and complexity metrics have. While complexity metrics are focused on a single code artifact (e.g. class), network metrics target the interaction between code artifacts.

2.3 Mining Software Repositories

Parts of the contents of this section have been published in Herzig and Zeller [70].

Software repositories record much of the activity around a software project. By *mining* these repositories *automatically*, you can obtain lots of initial evidence about your product—evidence that already is worth in itself, but which may also pave the path toward further experiments and further insights. During software development, programmers routinely produce and collect lots of data, all of which can be accessed and analyzed automatically:

- The *source code* for a product. This is the most important input to any analysis, as it provides *locations* (files, units, classes, components, etc.) that can be associated with various product or process factors.
- Collecting data on the execution of the software provides *profiles*, identifying frequently executed code parts and which parts were not.
- Documentation fragments such as *design documents* or *requirement documents* provide important features that explain why code looks the way it does or that explain why specific features have been implemented.
- The resulting software can be analyzed statically, providing features such as *complexity metrics* (see Section 2.2.1) or dependencies (e.g. see Section 2.2.3).
- *Version archives* (see Section 2.1.1) record the changes made to the product, including who, when, where, and why.

- To map problems to locations, one can use *issue tracking systems* (see Section 2.1.4) that describe problems that occurred and track their life cycles.
- Finally, *social data* can help to understand the developer team background behind the software product. With such data, for instance, it is possible to determine how much effort maps to individual tasks or locations, or how individual groups contribute to changes and to errors.

From a researchers point of view, the advantage of accessing these data sources is that they are *unbiased*—they record changes, problems, and other events at the moment they happen and with a realistic perspective that directly reflects the activities of the developers dealing with them. On the other hand, the data also may be *noisy*, *incomplete*, or even incorrect, which is why special steps are required before analysis.

Historic data for a software project is preserved though many different activities and many different systems. In order to extract and learn from these histories of a software project, accessing these resources is primary goal. For many open source systems (e.g. ECLIPSE or RHINO), most of these resources are publicly available and can be accessed easily. But each software repository is different, and so are the mining steps necessary to extract the relevant data points. Still, most mining tools and setups share a common procedure. This section introduces those common procedures and practices that are used or targeted by this thesis. The discussion is limited on two main subjects of mining software repositories: version control and issue tracking systems and how to connect these two independent repositories. Some of the experiments described in this thesis might also make use of additional repositories, procedures, and assumptions that will be explained in the corresponding sections describing the experimental setups.

2.3.1 Mining Version Archives

Version control systems maintain *change logs* for each change set—a set of simultaneously applied source code changes. These change logs contain information about the authors, the time of the change, the files that changed, how these files changed, and the commit message of the change set. Thus, parsing these code change logs provides the information about all source code changes ever applied to the software project.

There exists a wide range of approaches that mine version archives covering a wide range of purposes and application targets. German and Mockus [53] were among the first to present a tool called *SoftChange* that extracts and summarizes information from version archives. Since then, version archives have been used to build many different recommendation and prediction tools. Mockus and Votta [101] analyzed log messages using word frequencies and keyword classification techniques to determine the purpose of code changes applied to version archives. Giger et al. [57] used fine-grained source code changes (SCC) to predict whether a certain type of SCC affect a source file.

Version control systems allow to detect fine-grained logical couplings between classes [51], files and functions [149], code clones [52], and to detect patterns code artifacts frequently changing together [146, 154]. Hassan et al. [66] used version archives to assess the effectiveness of change propagation tools.

The number of research projects relying on version archives is endless and the small subset of presented approaches is far from being complete. The ever growing number of mining projects using different data layouts lead to exchange languages and data repositories that allow sharing and reusing existing data mining sets [24, 88]. There exists a wide range of approaches and studies combining version archives and bug databases. These papers are discussed in Section 2.3.4.

2.3.2 Dependencies between code changes

The difference between two (consecutive) source code revisions can be determined using a simple diff algorithm. But identifying the consequences of code changes and thus their dependencies between each other requires more detailed analysis. A code change adding a method call to a defined method (e.g. in Figure 2.1 `list.isEmpty()` on a collection) depends on the code change that added the method definition. But there exist also more complex dependencies. Gall et al. [50] were the first to examine version archives to detect logical couplings between program modules. Stoerzer et al. [132] determined *failure-inducing changes* using dependencies between code changes based on change location. Change sets depend on those change sets that previously changed the same or a subset of code lines. Going one step further, one can use the concept of *failure-inducing changes* to identify and extract code changes that can be applied to version archives without causing existing tests to fail [144]. Association rule mining techniques can be used to detect *coupled changes* and to automatically suggest and predict likely further changes [77, 127, 146, 154]. Fluri et al. [49] extracted hierarchically structured changes on statement level along with change type information. Later, they used their own tool to describe development activities using change patterns derived from change type combinations [48]. On method level, Kim et al. [85] presented an approach that represents structural changes as a set of high-level change rules, automatically infers likely change rules, and deter mines method-level matches based on these rules. Change couplings not caused by source code changes cannot be detected by program analysis and require structural change information to be detected [47]. To allow the detection of coupled changes over multiple change sets, Canfora et al. [29] used a sliding window approach to detect change couplings occurring within certain time frames across multiple change sets but without detecting structural dependencies between these change sets.

Impact analysis techniques such as CoverageImpact [113] and PathImpact [91] determine the immediate impact of code changes on program executions. Ren et al. [121] presented a tool to decompose program version differences into sets of atomic changes and to report test cases, whose execution behavior may have been changed. However, these approaches only consider the immediate impact of code changes on program structure and behavior. But these approaches assume that the software project can be completely compiled and executed. Consequently, these approaches are not applicable to all revisions of a project history that cannot be compiled.

To model dependencies between changes and change sets, Brudaru and Zeller [27] introduced the concept of *change genealogies*. A change genealogy is a directed graph structure to model dependencies between change sets. Change genealogies span the complete software project lifetime and allow reasoning about the impact a particular change set had on other, later applied change sets. German et al. [54] used the similar concept of *change impact graphs* to identify those code changes that influenced the reported location of a failure. But they resolved change dependencies backwards using a given starting point in the source code to identify possible causes for a bug or failure to occur. Alam et al. [5] used the concept of change dependency graphs [54] to examine how changes build on each other over time. The authors showed that dependencies between changes vary across different projects and that changes build on top of new code—instead on old stable code—are more defect prone.

2.3.3 Mining Issue Databases

Issue databases are a key factor of software maintenance containing a large proportion of problems and issues related to the software project. They contain the issue history of a software product. Learning from past issues is one of the key benefits of mining bug databases. Thus, many mining approaches are based on bug databases in some way, either as standalone artifact or as a combination between bug reports and code changes. It is important to realize that not every bug found in source code will be reported using an issue report. Many pre-release issues—issues that arise during development and intermediate testing—might get fixed unrecorded. Nevertheless, the larger fraction containing user issues and the most severe issues will be documented using issue reports.

Bug reports have been used to tackle a wide range of topics and problem sets. Among others, topics include automatic assignment of bug reports to developers [9, 31, 60], assigning locations to issues [30] using structural information [17], predicting and estimating effort for fixing bug reports [7, 19, 55, 95, 141, 147], bug triage [10, 135], and categorizing bug reports automatically [8, 130]. The quality of issue reports is a frequent discussed topic [14, 15, 59, 75]. Many of these studies show that bug reports often contain too little or too incomplete information in order to reproduce and fix them. It is possible to automatically detect bug report duplicates [16, 124, 138] that, when combined, might fill information gaps that prevent bug report fixes.

Antoniol et al. [8] showed that a significant number of bug reports refer to maintenance tasks, which are not of corrective nature. This finding implies that there exist a significant number of issue reports marked as source code defect although they were resolved and “fixed” without applying a source code path that fixed a code defect. Although their work had very little impact, Antoniol et al. [8] reported a highly problematic fact. Nearly all mining approaches rely on the fact that reported bugs are indeed documenting corrective maintenance tasks. We will address this issue in Chapter 4.

2.3.4 Mapping Artifacts across Repositories

In general, the more software repositories at hand, the more empirical data can be mined and the more evidence can be used to learn from the past. But having multiple software repositories at hand the more important to connect the artifacts in these repositories with the artifacts in the other repositories. As an example, combining bug reports to change sets opens the possibility to map bug fixes to individual code artifacts. Counting the number of fixed bug reports whose patch touched a specific file is frequently used to determine the quality of the source code. In most cases, bug report will not contain the applied source code changes. To detect which files were changed in order to fix the reported issue requires the version control system that contains the detailed changes. If both systems are not integrated or connected to each other mapping the artifacts to each other is a mining task.

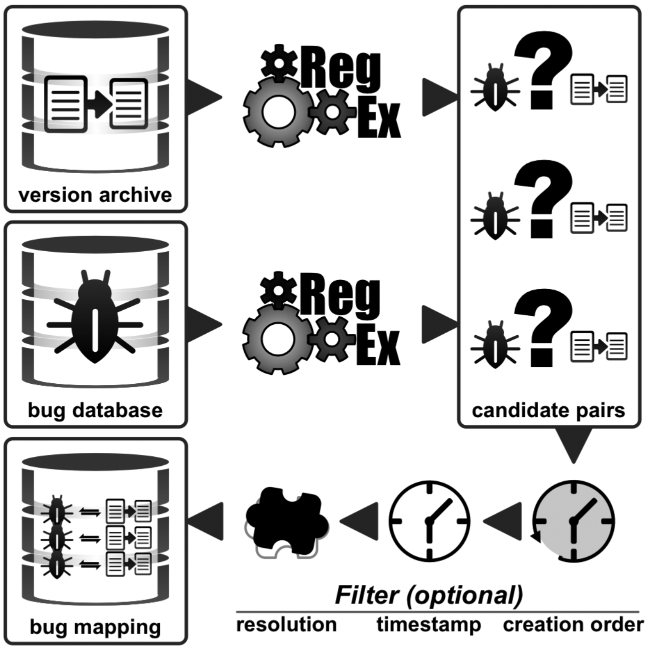


Figure 2.2: Mapping issue reports to code changes. Regular expressions used to identify issue report and change set references (e.g. `[bug|issue|fixed]:?\s*#?\s?(\d+)`) might differ from project to project. The number of used filters and their thresholds depend on the project.

One of the most important relations between software repositories is the connection issue tracking systems and version control systems. Fischer et al. [45] and Čubranić and Murphy [136] were among the first to search for references between code changes in version control systems and bug reports contained in issue tracking systems. The approach is straightforward. Commit messages of change sets contain references to bug report identifiers (e.g. “Fixes bug id 7478” or simply “fixes #2367”). These links will then be further filtered based on their activity, authorship and report date. Other approaches use slightly different techniques, but most of them are based on the same basic principle [45, 63, 86, 153]. Lately, Murgia et al. [103] presented an approach based on a more general natural language processing which groups commit messages sharing the same text features. Giger et al. [55] used bug report attributes to estimate the time to fix the corresponding issue. This thesis will also use code quality data derived using mapping strategies. Figure 2.2 describes the approach used in this thesis to map issue reports to change sets.

Modern development environments such as `Jazz` [78] integrate multiple software archives into a single development tool. This helps developers to work in one uniform environment without switching tool contexts. But integrating or connecting software archives also helps software archive miners. Mapping artifacts based on user input instead of heuristics and filters reduces data noise and help to prevent bias [69].

2.3.5 Classifying Code Changes

Classifying code changes is common in mining version control systems. There exist different approaches classifying code changes with respect to various aspects. Mapping bug reports to change sets (see Section 2.3.4) allows efficient bug fix identification. But these mapping techniques can only identify change sets that contain bug report references in the commit message.

Research has also focused on classifying code changes according to their impact on program execution [32], software architecture [42, 92, 133, 142], and program execution [32]. Closely related is the approach of Fluri et al. who developed a framework capable of differentiating "between several types of changes on the method or class level" [46]. With their framework, the authors are able to assess the impact of a code change on other source code entities and whether the applied change set modifies the functionality of the software system or not. Although, the our code change classification approach described in Chapter 6 uses a very similar abstraction layer, the aim of our approach is it to classify code changes with respect to their purpose: is a change set a bug fix or is it adding a new feature?

Kim et al. [86] classified code changes with respect to the likelihood that the applied change set introduces new software defects. Although this approach limits the search space for defect prediction models drastically, it cannot identify bug fixing change sets. Within their approach, the authors themselves used a commit message based approach to identify bug fixing change sets.

Lately, Kawrykow and Robillard [84] presented an approach that classified code changes due to their purity. Their framework allows to identify so called non-essential changes—changes that are of "cosmetic nature, generally behavior-preserving, and unlikely to yield further insights into the roles of or relationships between the program entities they modify" [84].

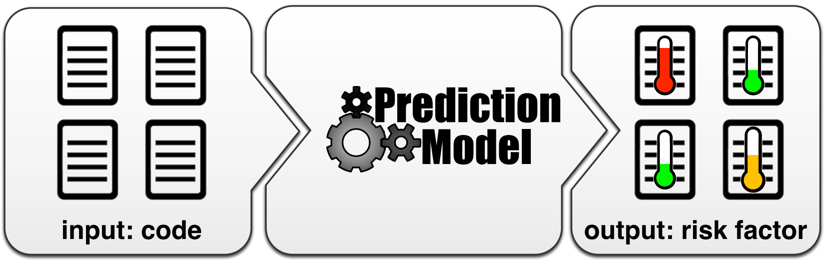


Figure 2.3: The operation mode of defect prediction models.

2.3.6 Defect Prediction Models

Data sets extracted from version control systems are frequently used to improve and support quality assurance tools, recommendation systems, and prediction models. The variety and range of approaches and tools using such data sets is huge and has gained a lot of attraction and activity over the past couple of years. For this thesis, approaches building and improving defect prediction models are of importance. This thesis presents techniques and approaches that can be used to further improve defect prediction models. Evaluation setups contained within this thesis will refer and make use of defect prediction models. Thus, describing the concept and fundamentals of defect prediction models shall provide fundamental information required to understand motivations, requirements, and evaluation setups. This section contains a discussion on basic concepts of defect prediction models and a rough review of past approaches predicting defects for software systems.

Defect prediction models aim to predict the number and sometimes the location of defects to be fixed in near future. Such systems can be used to allocate quality assurance resource that is deciding how much testing and reviewing effort should be applied to individual code artifacts. Defects and their impact are not equally distributed across the code base of a software project. Thus, distributing quality assurance equally over all code artifacts might be ineffective and might miss a significant number of defects that could have been found using a more sophisticated effort distribution. From a user's perspective, a defect prediction model should take a set of code artifacts and return a risk factor that indicates the likelihood that a given artifact contains a software defect (classification) or even more precisely a number of expected defects to be found within the code artifact (regression) as shown in Figure 2.3. Defect prediction models works as a black box. Supplied with a number of source files they return a risk factor for each

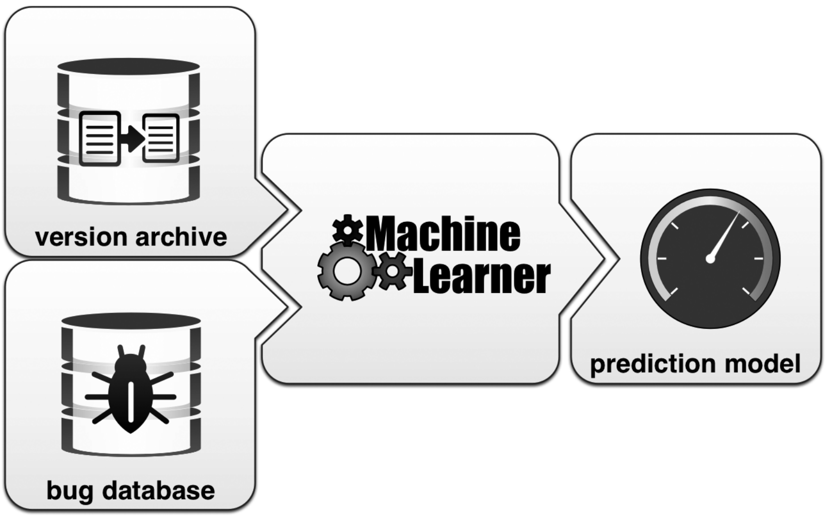
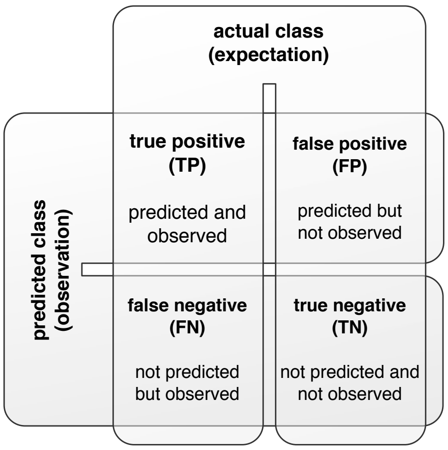


Figure 2.4: Training defect prediction models.

supplied input. Usually, defect prediction models are based on software metrics and historic data (e.g. past defects found per artifact). Defect-prone source files might be worth spending extra testing and reviewing effort on these artifacts. Complexity metrics and other software metrics are used to determine which features of the source code artifacts correlate with defect likelihood and thus can be used to check new code artifacts against harmful code features expressed by their corresponding software metrics.

To learn metric defect correlation, defect prediction models have to be trained on historic data sets as shown in Figure 2.4. Computing software metrics on a code base snapshot (see Section 2.2) and mapping past defect fixes to source code artifacts (see Section 2.3.4) can be used as defect prediction model training input. To model the relationship between the dependent variable (number of past fixes per artifact) and the explanatory variables (computed software metrics per artifact) can be done using various statistical analyses and machine learning techniques (e.g. linear regression, logistic regression, support vector machine).

Measuring the predictive power and prediction accuracy of a defect prediction model requires training and testing data sets. The training set will be used to model the relationship between past defects and code features while the testing set will be used to



$$\text{precision} = \frac{\text{TP}}{\text{TP+FP}}$$

$$\text{recall} = \frac{\text{TP}}{\text{TP+FN}}$$

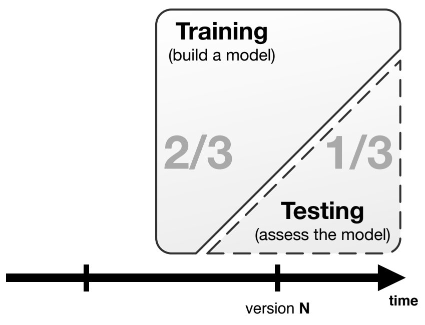
$$\text{accuracy} = \frac{\text{TP+FN}}{\text{TP+FP+TN+FN}}$$

Figure 2.5: Precision, recall, and accuracy.

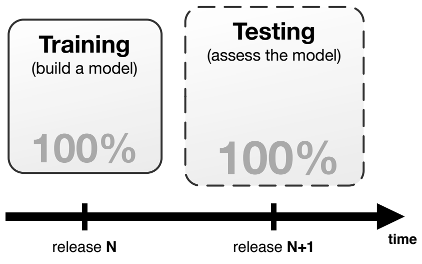
predict the risk factors for the code artifacts in the testing set. Comparing the prediction result with the known number of defects that were actually fixed allows to reason about the predictive accuracy. The higher the intersection of predicted and actual fixed defects the higher the prediction accuracy. To express the predictive power of a model in more detail one can use many different statistical measurements to express the fraction of correctly or falsely predicted entities. The most common set of measurements are *precision*, *recall*, and *accuracy*. These three measurements relate the number of *true positives* (TP), *false positives* (FP), *true negatives* (TN), and *false negatives* (FN) (see Figure 2.5).

Precision, recall, accuracy are fraction values between zero and one. A high precision means that the number of artifacts not predicted to be defect prone but observed to be defect prone is small. A high recall indicates a low number of false negatives—only few of those artifacts that were predicted as defect free where defect prone. Accuracy expresses the fraction of all correctly classified artifacts. There exist more prediction accuracy measurements that will be discussed where used in evaluation setups throughout this thesis.

Training and testing defect prediction models requires training and testing data sets. Figure 2.6 shows two common approaches to train and test defect prediction models. *Random splitting* a single data set into two subsets is frequently used if only one revision of a software project is available. The single data set is split into a training



(a) Random sampling or stratified random sampling splits one snapshot of a software project into 2/3 training and 1/3 testing parts.



(b) Using two releases or version of one or different project histories is closest to what can be deployed in the real world where past project data is used to identify defect-prone entities in on-going or future releases.

Figure 2.6: Generating training and testing data sets to build and assess prediction models.

set (usually containing two third of the original sets artifacts) and into a testing set (see Figure 2.6(a)). The intersection of training and testing set is empty while the union of training and testing data matches the original data set. *Stratified sampling* is used to preserve the original proportions of defect prone and non-defect prone entities in the training and testing sets.

Sampling data sets includes fuzziness: a single random sample can produce good results although the prediction model performs poorly on average. Thus, sampling is often combined with repeated holdout setups. Instead of splitting once, the data set gets repeatedly split into training and testing subsets and for each *cross-fold* or *holdout* precision, recall, and accuracy are recorded. The reported precision, recall, and accuracy values correspond to the mean values over the corresponding set of performance measures.

The alternative of splitting one revision of a software project apart is to use two revisions of the software code base (see Figure 2.6(b)). This method is commonly used to train and test prediction models based on releases. The earlier release serves as training set while the other, later revision is used to test the prediction model. Models are trained on revisions of different software projects. These *forward* or *cross-project* prediction setups are closest to what can be deployed in the real world where past project data is used to identify defect-prone entities in on-going or future releases.

Table 2.1: Overall defect prediction model accuracy using different software measures on WINDOWS VISTA. Content taken from [106] and [110].

Model	**Precision**	**Recall**
Change Bursts [110]	91.1%	92.0%
Organizational Structure [109]	86.2%	84.0%
Code Churn [104]	78.6%	79.9%
Code Complexity	79.3%	66.0%
Social network measures [22]	76.9%	70.5%
Dependencies [151]	74.4%	69.9%
Test Coverage [100]	83.8%	54.4%
Pre-Release Defects	73.8%	62.9%

Table 2.1 summarized the predictive power of post-release defect prediction models for WINDOWS VISTA categorized by the type of software metrics the models are based on. The differences in precision and recall measures show that different sets of software metrics heavily influence the prediction performance of the corresponding prediction model. Note that these are numbers for the MICROSOFT's WINDOWS VISTA software product, only. Switching to different software products in MICROSOFT or outside MICROSOFT might lead to different prediction performances and might also result in different rankings. As mentioned in Section 2.2, software metrics and their usefulness highly depend on a potentially large number of relevant context variables [12].

The number of studies and approaches related to defect prediction published is large and continues to grow. This section references only those approaches and studies that are closely related to this work. The given list of references is neither complete nor representative for the overall list of defect prediction models, their applications, and related approaches.

In 1996, Basili et al. [11] validated object-oriented design metrics as quality indicators and were among the first who empirically validated software metrics to be suitable to predict and estimate the quality of software products. In the same year, Ohlson and Alberg [112] presented one of the first empirical studies investigating the relationship between software metrics and code quality measured by the number of test failure reports. The authors also built a prediction model to identify the most defect prone entities. In 1999, Fenton and Neil [43] systematically reviewed and criticized approaches and studies related to defect prediction models and their theoretical and practical issues, while Briand et al. [26] performed an industrial case study confirming the correlation between complexity metrics and defects.

In 2000, Graves et al. predicted fault incidence using software change history. In their paper, the authors defined "code to be aged or decayed if its structure makes it unnecessarily difficult to understand or change and we measure the extent of decay by counting the number of faults in code in a period of time" [58]. Their paper was one of the first studies on defect prediction models based on change history and showed that process measures based on change history outperform complexity metrics when predicting fault rates. Denaro and Pezzè [41] showed that multivariate models can predict fault-proneness of code entities across different software packages. Ostrand et al. [114] showed that defect prediction models can be used to predict the number of faults for large industrial systems using four consecutive releases spanning a development time window of more than four years.

In 2005, Hassan and Holt [65] developed a dynamic fault prediction model based on changes and fix frequencies only. Such dynamic fault prediction systems can be used at any time and reflect changes to the top ten list of most defect prone entities as they occur without retraining a statistical model explicitly. Later, Kim et al. [89] extended this concept by adding not only frequently changed and fixed code entities to a so called "bug cache" but also closely related entities. This way, the bug cache containing 10% of all source code files represent up to 95% of faults fixed in the software product.

Nagappan et al. [104, 107, 108] distinguished between pre- and post-release defects and used historical in-process and static complexity metrics for prediction purposes (later verified by Holschuh et al. [74] using different industrial data). Schröter et al. [126] used past failure history and JAVA import statements to predict the number of defects for source files at design time; well before the actual code has been written and complexity metrics could be collected. In 2007, Zimmermann et al. published a study on defect prediction models targeting the open source software product ECLIPSE and published the data set listing the number of "pre- and post-release defects for every package and file in the Eclipse releases 2.0, 2.1, and 3.0" [153]. The data set also contains complexity metrics for files and packages.

Nagappan et al. [109] sowed the influence of organizational structure on software quality using a metric scheme to quantify organizational complexity. The authors showed that organizational metrics are one of the most effective defect prediction models for WINDOWS VISTA. Lately, Zimmermann and Nagappan [151] introduced network metrics in the field of defect prediction treating code entities as actors in communication or social media networks and method calls between these entities as links between actors. Later, Bird et al. [22] extended the set of network metrics by adding socio-technical network metrics to improve prediction accuracy. Tosun et al. [134] and Premraj and Herzig [119] replicated these studies on different open source data sets. To check weather defect prediction models also work across different projects, domains, and process models Zimmermann et al. [152] conducted a large-scale experiment on 12 real-world applications running 622 cross-project prediction models. The results were disillusioning. Only 3.4% of all prediction models actually worked. In 2011, Nagappan et al. [110] introduced the concept of change bursts—consecutive code changes on a set of files over a period of time. The authors (including the author of this thesis) showed that change bursts mark the best single defect predicting metric on WINDOWS VISTA. But the authors also showed that the very same metric fails for open source projects like ECLIPSE.

Lately, Giger et al. [56] used fine-grained source code changes to improve defect prediction models based on code churn metrics while Bird et al. identified "reasons that low-expertise developers make changes to components and showed that the removal of low- expertise contributions dramatically decreases the performance of contribution based defect prediction" [23].

2.4 Hypotheses, Noise, and Bias

The discussion of defect prediction models in the last section unveiled a couple of divergences. Defect prediction models highly depend on the underlying software project, the programming language and in particular on the underlying development process. Many promising approaches evaluated on industrial data sets performed poorly when evaluated on open source projects and vice versa. Cross project defect prediction models tend to work only under specific circumstances and the number and sets of software metrics performing well is changing not only from project to project but also differ between consecutive releases of the same software product. This section focuses on a very important but long ignored issue in data mining: underlying hypotheses, data noise, and the resulting bias. Mockus [99] and Liebchen and Shepperd [94] mentioned that data quality in empirical software engineering can be low and might impact the outcome of many empirical studies. Liebchen and Shepperd [94] found that only a tiny fraction of software engineering papers suggest data quality issues and their possible effect on their analysis results. Nguyen et al. [111] reported similar issues with studies investigating commercial software products, which usually follow more strict development guidelines compared to open source projects. This section contains a discussion over three hypotheses that are closely related with this thesis.

2.4.1 Hypothesis 1: We can map all bugs to code changes

We already discussed that mapping issue reports to change sets and thus mapping fixes to code entities is a key factor to measure the quality of the existing code base and thus key for the research domain of mining version archives. But how accurate can we map these artifacts? In 2009, Bird et al. [20] investigated bug and commit bias and their potential implications for defect prediction models. Their result showed the presence of systematic data noise and bias in several open source data sets affecting the performance of award winning defect prediction models. Later, Bird et al. [21] developed a framework allowing efficient manual inspections annotating mined data. But such manual inspection phases are rarely found in studies conducting defect prediction research. To ease the pain of manual inspection Wu et al. [145] developed an automatic link recovery algorithm to improve links between issue tracking systems and change sets in order to reduce data noise. Kim et al. [87] provided guidelines for acceptable noise levels and proposed a noise detection and elimination algorithm that identifies noised data instances.

2.4.2 Hypothesis 2: Bug reports are bug reports

Issue tracking systems contain many different types of issue reports: bug reports, feature requests, task descriptions, and requests for improvements. Most studies on defect prediction models consider issue tracking systems and their content as unbiased. But is every bug report a bug report? From a customer's point of view, this might be true since nearly every unexpected or undocumented behavior points to a defect. But does every bug report really lead to a corrective maintenance change set? In 2008, Antoniol et al. [8] showed that a potentially large fraction of issue reports in issue tracking systems of open source projects are wrongly classified. In the same paper, the authors provided an algorithm to automatically detect issue report types using the description and discussion within the issue report. But the authors did not show the exact extend of these misclassified issue reports and their potential threat on code quality models. In Chapter 4 we present the results of a manual inspection of over 7,000 issue reports and show that misclassified issue reports can severely impact code quality models.

2.4.3 Hypothesis 3: Code changes are atomic

For most approaches a mapping between change sets and bug reports is insufficient. To measure code quality, fixes need to be mapped to source entities such as source files or classes. For this purpose, issue reports have to be associated with those files that were changed by the associated and corresponding change set. The distinct number of associated issue reports per source files can then be used as quality measure for individual source entities.

But simply assigning the associated issue reports of a change set to the changed code entities assumes that all code entities were changed to resolve all assigned issues. We assume that either all code entities were changed to fix all assigned issues or that code changes resolving different issues are applied in separate change sets. Both is not the case. Kawrykow and Robillard [84] showed that up to 15% of all method updates were due to so called non-essential changes: "low-level code changes that are i) cosmetic in nature, ii) generally behavior- preserving, and iii) unlikely to yield further insights into the roles of or relationships between the program entities they modify." [84]. Thus, simply taking all code entities changed by a fixing change set as being fixed introduces imprecision. Up to 15% of these code entities were changed to perform cosmetic changes. Similar, what to do if a change set contains more than one bug fix? How many bugs do we assign to which code entity? What to do when multiple change sets are referencing the same issue report? Do we consider all changes as fixes or do we consider only the last one as fix and all others changes as being reverted? In Chapter 5 we discuss the issue of code changes serving multiple developer maintenance tasks and present an *untangling algorithm* that can be used to reduce the amount of bias introduced by non-atomic change sets.

2.5 Summary

In this chapter, we discussed the value of software archives that contain large amounts of artifacts that describe the software development process and daily developer activities. Mining these archives is a key component to learn from past failures and patterns and to use this historic knowledge to prevent future development issues. Combining version control systems and issue tracking systems allows to reason about past code quality. Software metrics have shown to be well suited for defect prediction purposes. The related work presented in this chapter shows that mining version archives has a large and active research community that gained more and more importance over the past decade.

But we also discussed unresolved, severe issues in the field of mining version archives. The number of studies showing the amount of data noise and bias is constantly growing. A lack of data integrity checks, which often require manual inspection, is threatening the community to produce unreliable algorithms and mining models. The number of solutions to reduce data noise and bias is limited and so are the studies measuring the impact of noise and bias on existing mining models.

Chapter 3

Change Genealogies

Parts of the contents of this chapter have been published in Herzig [73] and Herzig and Zeller [71].

3.1 Introduction

Software development is an incremental process that uses earlier stages of a software product to build newer versions. During software development, source code is added, changed and removed. Directly after applying a code change, the software structure and execution behavior may have changed [91, 113]. There exist many research studies to determine incomplete changes or further code changes that are likely to occur soon [29, 77, 85, 127, 146, 154].

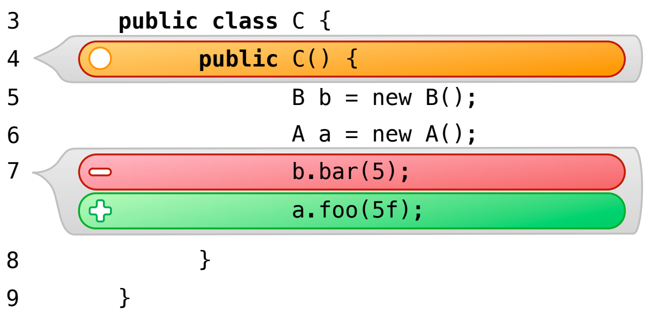
Research studies analyzing software histories have been mostly constrained to either space or time. Being constrained to space means to examine the evolution of single components, aggregating features over time. Being constrained to time means to examine which components were changed at a single moment in time, extracting co-changes from the resulting transactions. But we would like to reason over multiple components at multiple points in time. However, such reasoning requires a holistic view of all changes to all components.

Consider the following example: Your development team develops and maintains large-scale web applications with thousands of users. All web applications share a central authentication service that has been proven to be reliable, and secure. To further improve security, the team maintaining the authentication service decides to add a 2-way authentication option requiring users to enter a second key stemming from an additional device (e.g. PDA). The old authentication method, bypassing the newly introduced 2-way authentication process, gets deprecated to allow backward compatibility but to force all application teams to ensure a maximum of security. What will be the consequences of this design decision? How will this change impact the quality and stability of this project in the long term? How much development effort is necessary to change all references to the now deprecated authentication method? How much effort will it be to maintain two authentication APIs in parallel to allow backward compatibility? Influencing the development process over a period of time implies having influence on later decisions. Thus, measuring the long-term impact of a code change requires recording or reconstructing the dependencies between code changes. Change sets causing more dependent change sets have wider long term impact than others.

To allow such multi-dimensional reasoning and to measure long-term impact of code changes, we reused the theoretical concept of *change genealogies* [27] that model dependencies between change sets in a graph structure and built a practical change genealogy implementation modeling how changes influence and cause each other.

3.2 Source Code Change Dependencies

Brudaru and Zeller [27] proposed the theoretical concept of *change genealogies* that models dependencies between individual code changes. In this context, the term *code change* is too imprecise to allow any specific model to be built. Code changes can be captured in different levels of granularity: changes to tokens, changes to code lines, changes to methods, changes to files, etc. The lower the level of granularity the higher the number of entities and dependencies to be modeled, but the more precise the dependency model. On the other hand, choosing a low level of granularity requires very detailed and exhaustive code analysis techniques to detect dependencies. The lower the level of granularity the lower the context description making it harder to determine code changes touching the same change context.



```
public class C {
        public C() {
                B b = new B();
                A a = new A();
                b.bar(5);
                a.foo(5f);
        }
}
```

Figure 3.1: Sample diff output. Deleted method calls (DC) and definitions (DD) are marked red. Added method calls (AC) and definitions (AD) are marked green. Modifications applied to method definitions (MD) are marked orange. The diff output corresponds to table cell of column CS_4 and row $File_3$ shown in Figure 3.2. It also corresponds to the genealogy vertex CS_4 shown in Figure 3.6.

Independent from granularity, we define the dependency between code changes as:

Definition (Change Dependency)**.** *Code change C_2 depends on code change C_1 if and only if change C_2 can be applied only if change C_1 was applied before.*

Dependencies between code changes are directed. Each dependency points from the dependent code change (C_2) to the code change that is required to be applied first (C_1). Detecting all possible dependencies between code changes requires full type and cross-reference resolution over the complete project history. As an example: applying the code change shown in Figure 3.1 depends on each code changes that added

- the deleted statement `b.bar(5)`,
- the type definitions of `class A` and `class B`,
- the statements declaring the variables `a`, `b`,
- the method definition of `A.foo(float)`.

In object-oriented code, many dependencies between code changes are caused by methods calling each other. Adding a method call to a method that does not exist would cause compilation errors. Thus, applying a change set adding a method call depends on the change set that last updated the method definition of the method that is called (either modified or created). We call code changes that add, modify, or delete method definitions and method calls *change operations*[1]. There are many other possible interactions between code changes that would model code change dependencies on a lower level of granularity (e.g. token or statements). But in this thesis, we will model only dependencies between change operations and thus model only dependencies on method level.

Detecting dependencies between change operations aggregates many dependencies of lower level granularity into a much smaller set of dependencies, relaxing the complexity and size of change genealogies as dependency models, but introducing imprecision. Dependencies caused by changes to bindings and names are only covered if method calls were added, modified, or deleted. In any other case, these dependencies are not detected and modeled by change genealogies used in this thesis. Analyzing object-oriented code, change operations represent code changes that are guaranteed to influence either the interface of a module (e.g. adding or removing a method) or the execution behavior of the module (e.g. adding new method calls). Thus, change operations are likely to be dependent on past changes or be required by future changes.

To compute change genealogies on a project history, we reduce all change sets of the project history to sets of change operations that added or deleted a method definition (AD, DD), modified a method body (MD), or added or deleted a method call (AC, DC). Using an example change set that applied the code change shown in Figure 3.1 we derive a set containing three change operations. One change operation that deletes the method call `b.bar(5)`, one change operation that adds the method call `a.foo(5f)`, and one change operation that modifies the method definition `C.C()` containing the other two change operations. This example shown in Figure 3.1 corresponds to the two change operations shown in Figure 3.2 for column CS_4 and row $File_3$ using the resolved type binding of the individual called classes. The MD representing the modification of `C.C()` is an *indirect* change operation caused by the DC and AC operations in the method body of `C.C()`. Indirect change operations are hidden in Figure 3.2.

The set of possible dependencies between change operations is limited and can be described using a set of change dependency rules that can also be used to implement a dependency detection algorithm. Each change operation to a method depends on the previous modification or deletion of the very same method.

[1]A more detailed description on change operations can be found in Section 2.1.3.

Table 3.1: Change dependency rule set defining valid dependencies between change operations adding, modifying, and deleting method definitions and method calls.

Rule	Description
$AD_{\mathcal{D}} \rightarrow DD_{\mathcal{D}}$	A change operation AD adding a method definition $\mathcal{D}$ might depend on a DD deleting a method with the same full-qualified path. (One can define a method only once.)
$MD_{\mathcal{D}} \rightarrow AD_{\mathcal{D}}$ $MD_{\mathcal{D}} \rightarrow MD_{\mathcal{D}}$	A change operation MD modifying a method Definition $\mathcal{D}$ depends on the last change operation to the definition $\mathcal{D}$ of the method (this can either be a modification MD to $\mathcal{D}$ or the creation AD of $\mathcal{D}$).
$DD_{\mathcal{D}} \rightarrow AD_{\mathcal{D}}$ $DD_{\mathcal{D}} \rightarrow MD_{\mathcal{D}}$	A deletion DD of a method definition $\mathcal{D}$ depends on either the initial AD that added method definition $\mathcal{D}$ or the last MD that modified $\mathcal{D}$ (if $\mathcal{D}$ got modified since its definition).
$AC_{\mathcal{D}} \rightarrow AD_{\mathcal{D}}$ $AC_{\mathcal{D}} \rightarrow MD_{\mathcal{D}}$	A change operation AC adding a call to method $\mathcal{D}$ depends on either the initial AD that added method definition $\mathcal{D}$ or the last MD that modified $\mathcal{D}$ (if $\mathcal{D}$ got modified since its definition).
$DC_{\mathcal{C}} \rightarrow AC_{\mathcal{C}}$	A change operation DC deleting a method call $\mathcal{C}$ depends on the initial AC that added method call.

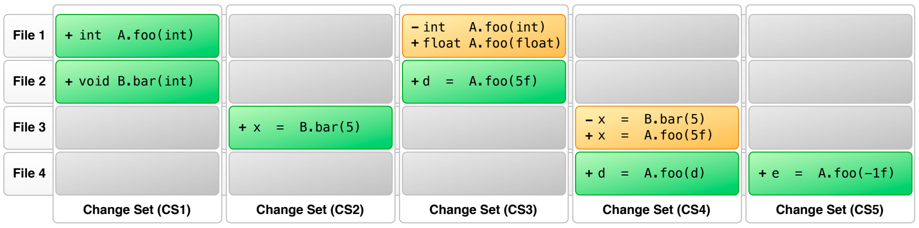


Figure 3.2: We characterize change sets by method calls and definitions added or deleted. Changes depend on each other based on the affected methods.

Change operations adding a method call depend on the change operation that added the called method definition or the change operation that changed the method definition. A change operation deleting a method call depends on the change operation that added the method call just deleted. The complete change dependency rule set also contains MD change operations can be found in Table 3.1. The change dependency rule set strictly follows the basic dependency definition. Method definitions are identified using their full-qualified name. Method calls are identified using the absolute position[2] of the method calling statement in the source code. Dependencies between change operations are not only directed but also labeled. The edge label contains the full-qualified name of the method definition or method call that caused the dependency (see Table 3.1 for detailed description). Although we limit the concept of change operations to method definitions and method calls in this thesis, the basic concept of dependencies between code changes is independent from the chosen level of change dependency granularity. When switching to more fine grained granularity levels (e.g. line or token based) the concept of change genealogies and all approaches relying on the given change dependency rules will remain valid and functional.

3.3 The Concept of Change Genealogies

The original concept of change genealogies as given by Brudaru and Zeller [27] models change dependencies using directed acyclic graphs. These graphs contain edges $C_2 \rightarrow C_1$ if and only if change C_2 can be applied only if change C_1 was applied before. This model matches our change dependency definition given in Section 3.2. But dependency graphs modeling dependencies between individual change operations will not be

[2]source file and character position within file

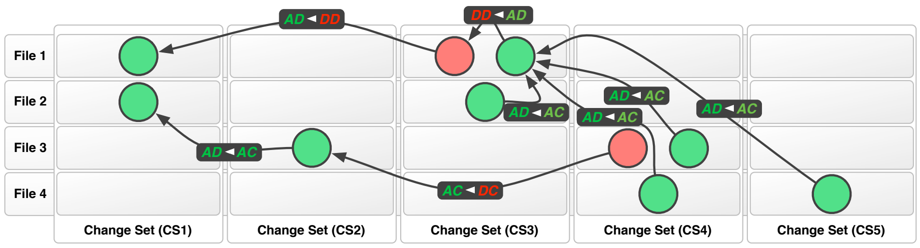


Figure 3.3: Change genealogy modeling sample history shown in Figure 3.2.

acyclic. Adding two method definitions that call each other will cause a dependency cycle. We will deal with that problem in Section 3.4. For now, we use the definition of change genealogies as give by Brudaru and Zeller [27] without requiring a genealogy graph to be acyclic.

The change genealogy graph structure consists of vertices, each corresponding to exactly one change operation. Edges represent a direct structural dependency between the corresponding change operations (see Section 3.2). The direction of a dependency edge corresponds to the direction of the dependency between source and target. Each dependency edge is annotated with a rationale—the change dependency rule that legitimates the dependency edge (see Table 3.1). If there exist multiple dependencies between the same pair of change operations, the rationale comprises them all. Change genealogies based on change operations are likely to contain cycles. Change operations that depend on each other and being applied simultaneously are occurring frequently.

Figure 3.3 shows a change genealogy graph that models all change operation dependencies extracted from our sample history shown in Figure 3.2. Each change operation shown as `diff` statement in Figure 3.2 is represented as graph vertex in Figure 3.3. Change sets on one file might be represented using multiple genealogy vertices as shown for CS_3 on $File_1$ in Figure 3.3. One genealogy vertex (left) for the removal of method definition `int A.foo(int)` and one genealogy vertex (right) for the addition of method definition `float A.foo(float)`.

3.3.1 Properties of Change Genealogies

Change genealogies are simple graph structures modeling structural dependencies between individual change operations. The various properties of the graph structure make change genealogies powerful.

Vertices represent temporal and spatial position. Each vertex of a change genealogy corresponds to a *change operation* once applied to the software project. Change operations are bound to a specific point in time—the timestamp of the change set that applied the change operation—and to a specific location—the code artifact the change operation was applied to. Having a two dimensional coordinate (temporal and spatial) each vertex does not only represent a change operation being applied to the source code, but also a historic event and its exact position in the two dimensional space.

Vertex annotations. Vertex annotations add important *context information*. Each vertex is annotated with the original chunk of changed lines explaining the change operation and with the timestamp, author, and identifier of the change set that applied the change operation, and with the full-qualified method, class, and file names that were affected by the change operation (see Figure 3.4). Using vertex annotations it is possible to retrieve details about the context in which the change operation was applied. Each vertex annotation references the original change set in the version control system allowing to map genealogy vertices to other artifacts such as issue reports as discussed in Section 2.3.4. For reasons of simplicity, vertex annotations are not shown in Figure 3.3.

Edge annotations. Edge annotations add *rationales* to each modeled dependency. The rational identifies the change dependency rule that legitimates the dependency edge as shown in Figure 3.3. If there exist multiple dependencies between the same pair of change operations, the rationale comprises them all. The full- qualified names of the method definitions and method calls depending on each other can be retrieved from the corresponding vertex annotations. Change genealogy edge annotations enable the user to ignore single or multiple change dependency rules for analysis purposes. If an analysis of change dependencies shall ignore all dependencies based on method calls or modification change operations, one simply ignores edges annotated with corresponding change dependency rules when traversing the change genealogy graph.

Edges span periods of time. The source and target change operations of a dependency edge may be applied at *different points in time*. Thus, change genealogy edges span arbitrary long periods of time. The vertex annotations contain the timestamps at which source and target vertex were applied. Using this information the actual length of the time span can be derived. An important property of change genealogies is that the dependent change operations are either applied after or simultaneously with the dependency edge target change operations—the future cannot influence the past.

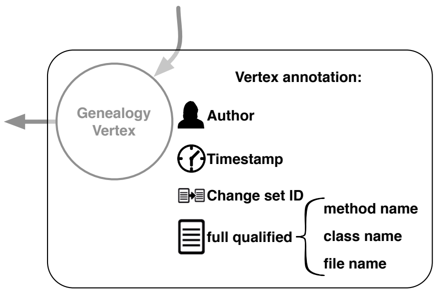


Figure 3.4: Example change genealogy vertex annotation showing the annotation content that is attached to every change genealogy node.

Spatial dimension. *The spatial dimension of a change genealogy* is given by the fact that different change operations might affect different source entities such as methods, classes, and files. Thus, change genealogy edges might link different space coordinates with each other.

Cycles. Change genealogies are likely to contain *dependency cycles*. Although the theoretical, original definition of change genealogies [27] did not allow dependency cycles, the change genealogy implementation used in this thesis is likely to contain dependency cycles. Two change operations applied in the same change set and depending on each other (e.g. adding two method definitions calling each other) will introduce a dependency cycle in the corresponding change genealogy. It is desirable to define change genealogies as directed acyclic graph since acyclic graphs would define a topological order on dependent change operations. We will deal with this problem later in Section 3.4.

3.4 Change Genealogies Layers

The basic concept of change genealogies as discussed in Section 3.3 models dependencies between single change operations. Although this level of granularity already ignores more detailed dependencies between modifications of single statements or tokens, there exist cases in which dependencies between change operations are already too fine grained. Analyzing the change dependencies between different developers requires dependency analysis between change sets. Change operations in the same

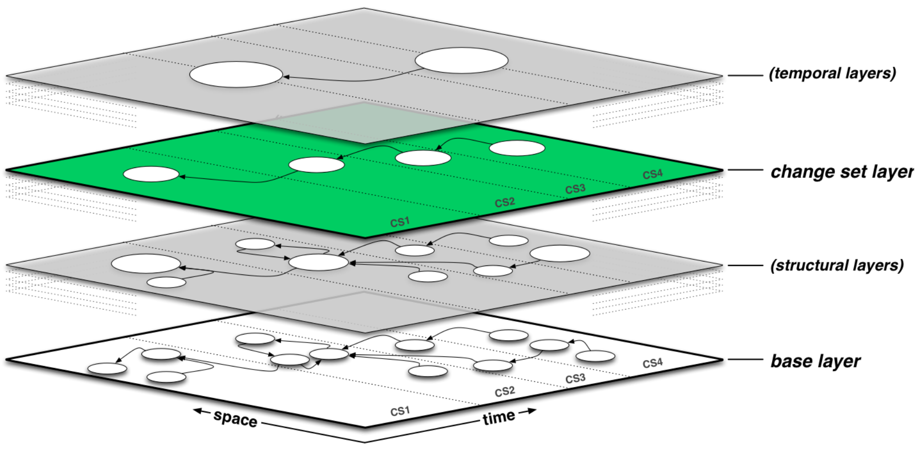


Figure 3.5: Change genealogy layers.

change set are applied by the same developer and thus can be combined. To support higher-level change genealogy analyses, the change genealogy framework presented in this thesis can generate different layers on top of the basic change genealogy layer modeling change operation dependencies. For this purpose, a partition of change genealogy vertices is generated on the fly without the need to recompute vertex dependencies. Dependency edges connecting two change operations in the same higher-level partition get dropped. Dependencies between two change operations that are contained by different partitions will be represented by a dependency edge between the two corresponding change operation partitions. Figure 3.5 shows the concept of change genealogy layers graphically. The higher the change genealogy layer, the higher the dependency abstraction. One can compare layers with zoom levels on street maps. The lower the zoom factor, the lower the details presented on the map. Although tiny streets are not visible anymore, connections between major cities remain visible. The developed change genealogy framework allows user defined partition algorithms to create arbitrary user defined partition layers. We divide change genealogy layers into two main categories: *structural layers* and *temporal layers*. Change genealogies remain directed, independent from the layer and the partition type. But partitioning the change genealogy graph structure might add additional graph properties to the change genealogy layer. While every basic change genealogy graph structure is likely to contain path cycles, there exist graph partitions that are guaranteed to be acyclic.

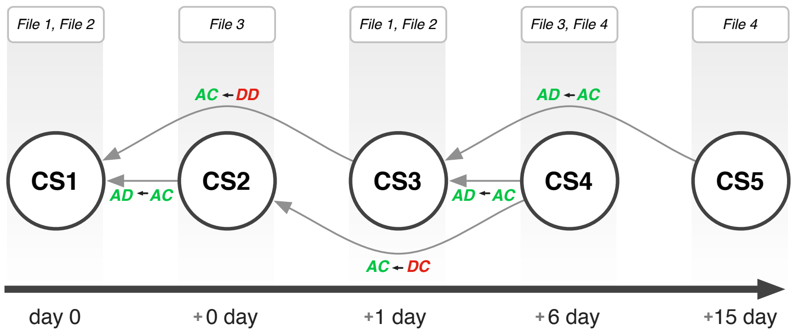


Figure 3.6: Change set layer of the change genealogy shown in Figure 3.3.

Structural layers are based on partitions based on the structural dependency between change operations within the same change set. A useful structural layer might be derived by grouping change operations within a single change set such that each change operation group contains changes applied to the same source artifact (e.g. source files). Using such a structural layer raises the level of granularity from method definitions and method calls to source file level. Temporal layers create change operation partitions based on their temporal dependencies. As indicated in Figure 3.5, the graph partitions in these layers span multiple change sets. This way one can model dependencies between different development phases (e.g. between weeks or between development and testing phase).

Change Set Layer

One of the important temporal change genealogy layers is the so called *change set layer*. The layer corresponds to a change genealogy in which vertices represent different change sets applied to the version control system. Dependency edges model dependencies between individual change sets—dependencies between change operations that were applied at different points in time. Figure 3.6 shows the change set change genealogy layer of our initial example history shown in Figure 3.2. The presented change set layer is based on the change genealogy presented in Figure 3.3 and simply aggregates vertices and dependencies to the change set level. As shown, at the change set level each vertex corresponds to exactly one change set of the project history. A change set CS_j depends on a change set CS_i if and only if at least one of the

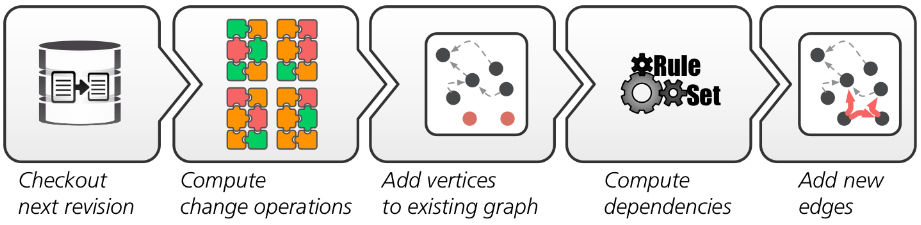


Figure 3.7: Overview over the change genealogy computation process. This overview corresponds to the overall algorithm as shown in Algorithm 3.1.

change operations applied by CS_j depends on at least one of the change operations applied by CS_i. Edge annotations at the change set layer contain all edge annotations of those edges that reach from one change set partition to the other. At this layer, different vertices were applied at different points in time allowing a strict temporal order $\mathcal{O}_{temp}$ of change genealogy vertices. It is natural that change operations cannot depend on change operations that have not been applied at that time. There exist only dependency edges $v_i \leftarrow v_j$ where $i \prec j \in \mathcal{O}_{temp}$. Thus, $\mathcal{O}_{temp}$ guarantees the change genealogy layer to be *acyclic*.

3.5 Extracting Change Genealogies

The computation of change genealogies is straightforward but requires high computational effort. Change genealogies are based on change operations. Constructing change genealogies requires the computation of change operations for the entire given version history. To compute change operations for an individual change set we have to checkout and compile the corresponding change operation and its precedent change operation. Depending on the length of the project history and the size of each individual change operation this process might take days. The overall process of generating change genealogies is shown in Figure 3.7 and described in more detail in Algorithm 3.1 using pseudo code. The process diagram from Figure 3.7 is repeated for every change set in the version archive.

Line 2 of Algorithm 3.1 calls a function that generates the change operations for a particular change set. This function is described in Section 3.5.1. Our overall algorithm then iterates over all change operations of the current change set to search for added

```
Input: Version Archive SCM
foreach change set $CS_i \in$ SCM do
    $OPS_i$ = getChangeOperations ($CS_i$);          // See Section 3.5.1
    foreach $OP_i \in OPS_i$ do
        addVertexToGenealogy ($OP_i$);           // Simply add vertices
    end
    addEdgesToGenealogy ($OPS_i$);               // See Section 3.5.2
end
```

Algorithm 3.1: Algorithm to compute change genealogies.

(AC) and deleted (DC) method calls. This is necessary since AC and DC operations potentially change a method definition by modifying the method body. Thus, the algorithm fetches the surrounding method definition of each AC and DC change operation and adds the newly created change operation to the list of change operations for the current change set. Once all change operations are computed, we add a vertex for each change operation to the change genealogy data structure. The procedure describing the usage of change dependency rules (see Section 3.2) and the computation of change genealogy edges is described in Section 3.5.2.

Algorithms presented in this chapter assume can only operate on single software development branches. Change genealogies covering multiple branches are not supported and should in general be handled with care. Multiple branches can contain multiple versions of method definitions with the same full-qualified names referring to different method definitions.

3.5.1 Computing Change Operations

This section contains a detailed description of how to extract change operations from version archives. As discussed in the previous section, change operations are extracted based on change sets. For each change set the algorithm will return a set of change operations that were applied within the change set. All change operations discussed in this section are based on method definitions and method calls declared in object-oriented source code. Still, the general approach is also applicable for other levels of source code entity granularity as well as for non object-oriented code requiring adaptations and many more special cases.

```
Data: $Def_{CS_{i-1}} = \{\text{method definitions} \in CS_{i-1}\}$
Data: $Call_{CS_{i-1}} = \{\text{method calls} \in CS_{i-1}\}$

Input: $CS_i$
Output: $(DD_i, MD_i, AD_i, DC_i, AC_i)$ ;       // Change operations of $CS_i$

1  $F_{\text{modified}}$ = checkout($CS_i$) ;                          // modified files
2  $DD_i = \emptyset$; $MD_i = \emptyset$; $AD_i = \emptyset$; $DC_i = \emptyset$; $AC_i = \emptyset$;

3  foreach $f \in F_{\text{modified}}$ do
       // method definitions and calls in f
4      $(Def_{(f,CS_i)}, Call_{(f,CS_i)})$ = PPA($f$) ;                // using PPA [37]
5      $AD_i = AD_i \cup (Def_{(f,CS_i)} \setminus Def_{CS_{i-1}})$;
6      $DD_i = DD_i \cup (Def_{CS_{i-1}} \setminus Def_{(f,CS_i)})$;

       // we identify calls using their code position that might
       // be shifted (no details here).
7      $AC_i = AC_i \cup (Call_{(f,CS_i)} \setminus Call_{CS_{i-1}})$;
8      $DC_i = DC_i \cup (Call_{CS_{i-1}} \setminus Call_{(f,CS_i)})$;

       // check for implicit MD caused by
       // added and deleted method calls
9      foreach $OP_{call} \in \{AC_i \cup DC_i\}$ do
           // get method definition modified by call operation
10         $\mathcal{D}$ = getSurroundingMethodDef($OP_{call}$);
11         if $\{DD_i \cup AD_i \cup MD_i\}$ contains no operation on $\mathcal{D}$ then
12             $MD_i = MD_i \cup$ {createMD($\mathcal{D}$)};
13         end
14     end
15 end
```

Algorithm 3.2: Algorithm to compute change operations based on change sets.

The algorithm to compute the change operations for a given change set (see Algorithm 3.2 on page 44) takes a set of defined method definitions and method calls as input (in Algorithm 3.2 it is a map associating method definitions and method calls to change operations but for this algorithm a simple set would suffice). Computing change operations for a series of consecutive change sets you can pass the return value of Algorithm 3.2 for CS_i as input for the algorithm computing change operations for CS_{i+1}. For the first change set of the series CS_{start} one has to compute this set of present method definitions based on the source code revision $CS_{start-1}$. If CS_{start} equals the very first revision in the project's history, the set can be left empty since no source code was present before CS_{start}.

Algorithm 3.2 hides the details of source code parsing and is not optimized to keep the pseudo code as simple as possible. In general, we try to compile each source code file using the *partial program analysis* tool (PPA) for Java [37] that is based on the Eclipse JDT[3] framework. Provided with the source code, the PPA tool provides a abstract binding tree (ABT) corresponding to this source code. We identify method definitions and calls by simply traversing the ABT returned by PPA using the standard JDT visitor pattern framework. We ignore type bindings that cannot be identified by PPA (e.g. dynamically dispatched type bindings).

The full-qualified Java method definition signature can be used as a primary key to identify every Java method definition. Multiple methods having the exact same full-qualified signature are prohibited by the Java language standard. Thus, identifying added and deleted method definition change operations can be done using set operations. This is different from Java method calls (line 3 to line 15 in Algorithm 3.2). For method calls we have to determine pairs of old and new method call locations. Since code lines were added before individual method calls we have to compute offsets to find for each method call location the corresponding method call location of the previous revision. Those method calls that have no corresponding location are newly added. For offset computation we use GNU diff blocks.

The output of this change operation computation algorithm presented in Algorithm 3.2 can be directly used to add change genealogy vertices. For each computed change operation we add one vertex to the base change genealogy layer. The algorithm used to add change genealogy vertices is discussed in the next section.

[3]http://www.eclipse.org/jdt/

3.5.2 Computing Genealogy Edges

In the previous section we discussed how to compute change operations corresponding to line 2 in Algorithm 3.1. Each of the change operations computed will finally be added (see line 4 in Algorithm 3.1). In this section, we will go through the algorithm that uses the change dependency rules discussed in Section 3.2 to add edges to the change genealogy under construction. Before executing line 6 in Algorithm 3.1 the change genealogy graph structure contains only change operations applied in this change set but no edges to earlier applied change sets.

The pseudo code algorithm adding genealogy edges is presented in Algorithm 3.3 and mainly consists of *switch* statements to handle the appropriate change dependency rules. The algorithm makes use of the fact that Algorithm 3.1 iterates over change sets in topological order. This way, we can maintain so called *operation stacks* that contain one stack of change operations for each known method definition and method call instance. Using the operation stacks, we can quickly identify those change operations that modified a method definition or method call previously (also in topological order). The newly processed change operations get pushed onto the corresponding change operation stack. Each added edge is labeled with the change dependency rule that caused the edge to be added. Not shown in Algorithm 3.3 is the case for duplicate edges. If an edge between two change operations already exists, we simply update the label of the existing edge by adding the change dependency rule if not already present. The most important part of the edge computation algorithm is the change operation sorting (see line 1 in Algorithm 3.3). Sorting the change operations is crucial since it ensures correct inner change set dependencies. If a change operation adding a method call to a method *m* is processed before the change operation adding *m* in the same change set the method stacks would not contain the AD change operation adding *m* and thus we would miss important inner change set dependencies. The total order of change operations is bound to change operations in the same change set. In one change set, each method definition and call can only be assigned to one change operation.

With the termination of this algorithm we successfully completed one computation process (see Figure 3.7) for one change set. This sequence of operations is then repeated for all remaining change sets in topological order. The algorithm can be terminated after each change set iteration without compromising the underlying change genealogy structure. Adding more change sets to an already existing change genealogy requires reconstruction (e.g. serialization and restore) of the operation stacks. Computing a complete change genealogy typically takes hours but highly depends on the history and code size of the project under analysis. Adding a small set of change sets to an existing change genealogy (e.g. a development week) usually takes minutes.

```
Input: $OPS_i$
Data: $Def_{\mathsf{Map}}$ = a map from method definitions to a list of change operations
      last modifying the definition.
Data: $Call_{\mathsf{Map}}$ = a map from method call location to a list of change operations
      last modifying the call.
// Sort change operations by priority:
// DD ≺ MD ≺ AD ≺ DC ≺ AC
$OPS_i$ = sortChangeOPs($OPS_i$);

foreach $op$ in $OPS_i$ do
    switch the type of change applied by op do
        case AD
            $\mathcal{D}$ = full-qualified name of added defintion;
            $Def_{pref} = Def_{\mathsf{Map}}$ [$\mathcal{D}$][end] ;          // might also be a DD
            addDependency($Def_{pref} \leftarrow op$);
            $Def_{\mathsf{Map}}$ [$\mathcal{D}$][end + 1] = $op$
        case MD
        case DD
        case AC
            $\mathcal{D}$ = full-qualified name of added defintion;
            $Def_{pref} = Def_{\mathsf{Map}}$ [$\mathcal{D}$][end] ;          // must not be a DD
            addDependency($Def_{pref} \leftarrow op$);
            $Def_{\mathsf{Map}}$ [$\mathcal{D}$][end + 1] = $op$
        case DC
            $\mathcal{L}$ = location of deleted method call;
            $Call_{pref} = Call_{\mathsf{Map}}$ [$\mathcal{L}$][end] ;        // must not be a DC
            addDependency($Call_{pref} \leftarrow op$);
            $Call_{\mathsf{Map}}$ [$\mathcal{L}$][end + 1] = $op$

    endsw
end
```

Algorithm 3.3: Algorithm to compute change genealogy edges given a set of change operations of an change set.

3.5.3 Change Genealogy Layers as Intelligent Wrappers

In Section 3.4 we discussed the concept of change genealogy layers. Change genealogy layers are read-only layers that can be used to inspect the underlying change genealogy from different perspectives. Change genealogy layers cannot be used to modify the underlying change genealogy. This way, we ensure that the base change genealogy contains edges between individual change operations. If we would allow adding edges between two change sets, we could not determine the corresponding change operations that caused the edge to exist and thus would destroy the accuracy of other change genealogy layers. Each change genealogy layer is operating on the basic change genealogy layer and its underlying graph database translating read requests (e.g. get all dependent nodes) into read requests sent to the basic change genealogy layer. In similar fashion, the response value returned by the basic change genealogy layer is transformed or aggregated into a response value that matches the layer partition strategy. The algorithms 3.4 and 3.5 show pseudocode translating read requests into base layer read requests. Algorithm 3.4 shows the pseudo code to check whether the change set change genealogy layer contains an edge between two change sets. Algorithm 3.5 shows pseudo code for getting all dependent change sets of a given change set. Due to the fact that change genealogy layers are read only layers implementing a new change genealogy layer means to implement a single class interface.

3.6 Change Genealogy Assumptions

Like many other mining approaches, change genealogies depend on the development process used when the analyzed software was implemented. Violations of these assumptions have to be considered to bias approaches based on such assumptions. In this section we will discuss the most threatening assumption that we made when constructing change genealogies and in particular when defining the default change set change genealogy layers.

When lifting the level of granularity from basic change operations to change sets, we combine all basic change operations change genealogy vertices that are applied simultaneously to the version control system into one higher level change set change genealogy vertex. Consequently we assumed that all code changes applied in a change set depend on each other (see Section 2.4.3). But this is only true if we can assume that all code changes applied in the same change set serve the same developer maintenance

```
Data: $CG_{basic}$ = basic underlying change genealogy model
Input: $CL_{from}$ = change set that shall be source of edge.
Input: $CL_{to}$ = change set that shall be target of edge.

foreach $CO_{from} \in$ getAppliedChangeOperations($CL_{from}$) do
    foreach $CO_{to} \in$ getAppliedChangeOperations($CL_{to}$) do
        if edge($CO_{from}$,$CO_{to}$) $\in CG_{basic}$ then
            return true
        end
    end
end
return false
```

Algorithm 3.4: Change set change genealogy layer algorithm to check if an edge between two change sets exist.

```
Data: $CG_{basic}$ = basic underlying change genealogy model
Input: $CL_{in}$ = change set to get dependent change sets for

result = $\emptyset$;
foreach $CO_{in} \in$ getAppliedChangeOperations($CL_{in}$) do
    foreach {edge($CO_{dep}$,$CO_{in}$) $\in CG_{basic}$ } do
        if $CO_{dep} \notin CL_{in}$ then
            result = result $\cup \{CO_{dep}\}$
        end
    end
end
return result;
```

Algorithm 3.5: Change set change genealogy layer algorithm to get all dependent change sets of a given change set.

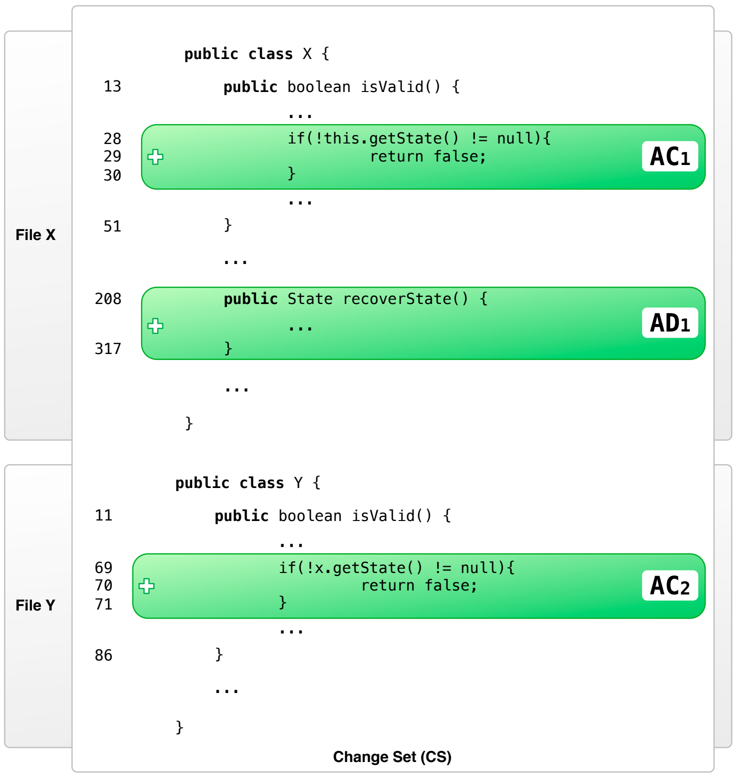
task. If the version control system contains *tangled* change sets, which serve multiple developer maintenance tasks, aggregating from the basic change set to any higher change genealogy layer is likely to add further imprecision and thus bias approaches based on change genealogies.

To explain how tangled change sets affect higher layers of change genealogies lets consider a typical example of tangled change sets. Figure 3.8 shows a real world example change set that changes two source files F_x and F_y. The change set adds one new method definition to class `X` (AD_1) and two method calls (AC_1 and AC_2) to class X and class Y. The commit message of the corresponding change set states: "Fixing issues #956 and #1072". Issue `#956` is a bug report while issue `#1072` requests a new feature. On the basic change genealogy layer that models dependencies between individual change operations this tangled change set causes no issues. But when using the higher order change set change genealogy layer, all three change operations will be in the same change genealogy partition and thus will be represented by a single change genealogy vertex. The corresponding vertex represents a feature implementation and a bug fix and combines the change set dependencies caused by the bug fix (AC_1 and AC_2) and the feature implementation (AD_1). Thus, the corresponding change set change genealogy layer models AD_1 to be dependent on the method definition of `X.getState()` and the change operations AC_1 and AC_2 depend on those method definitions called within the newly added method `X.recoverState()`.

A classification model that separates bug fixing from feature implementing change sets can classify such tangled change sets either as bug fix or as feature implementation. Independent from the decision of the classification model, the classification returned by the models has to be wrong. A single change set can only be classified as bug fixing or feature implementing but served both development purposes.

Tangled change sets also affect approaches predicting the long-term impact of code changes or predicting defects for source files. The wrongly combined dependencies of both developer maintenance tasks may indicate long-term cause effect chains (Chapter 7) that do not exist. Similar, larger tangled change sets being a composition of multiple trivial code changes can be interpreted as a single crucial code change influencing defect prediction models as described in Chapter 6.

In Chapter 5 we will investigate the impact of tangled code changes on other common mining approaches and provide an algorithm that can be used as preprocessing step to reduce the amount of data noise introduced by tangled change sets.



```
public class X {
    public boolean isValid() {
        ...
        if(!this.getState() != null){
            return false;
        }
        ...
    }
    ...
    public State recoverState() {
        ...
    }
    ...
}
```

```
public class Y {
    public boolean isValid() {
        ...
        if(!x.getState() != null){
            return false;
        }
        ...
    }
    ...
}
```

Figure 3.8: Tangled change set example.

3.7 Summary

Constant changes in requirements induce constant code changes. It is more than natural that code changes depend on each other. Understanding the complexity of software development activities and being able to track back decisions that formed the code base provides fundamental information that determines the quality of a change. Change genealogies model change operation dependencies using structural code dependencies that cannot be detected by standard mining techniques [30, 146, 149]. In general, change genealogies are not bound to model dependencies between change operations. It is possible to choose more fine grained dependency levels which then would require a much larger set of change dependency rules and require more detailed source code dependency analysis techniques. Change genealogies presented in this thesis are based on projects written in Java and use dependencies determined on method level. Adapting change genealogies to other object-oriented programming languages is possible but would require different source code dependency analysis tools. Switching from purely object-oriented code to functional programming languages or mixed languages such as C would require a definition of change dependency rules for code changes affecting functional language constructs and entities. Considering all vertex and edge properties, change genealogies model structural code change dependencies between individual change operations applied at different times and affecting different source code entities.

Chapter 4

It's not a bug. It's a feature.

Parts of the contents of this chapter have been published in Herzig et al. [68].

4.1 Introduction

In empirical software engineering, it has become commonplace to mine data from change and bug databases to detect where bugs have occurred in the past, or to predict where they will occur in the future. Later in Chapter 6, we will associate bug reports to code changes to mark change sets as bug fixes and to reason about the purpose of code changes and code quality. The accuracy of such measurements and predictions depends on the *quality of the data.* Therefore, mining software archives must take appropriate steps to assure data quality.

A general challenge in mining is to separate *bugs from non-bugs*. In a bug database, the majority of issue reports are classified as *bugs*—that is, requests for corrective code maintenance. However, an issue report may refer to "perfective and adaptive maintenance, refactoring, discussions, requests for help, and so on" [8]—that is, activities that are unrelated to errors in the code, and would therefore be classified in a non-bug category. If one wants to mine code history to locate or predict error prone code regions, one would therefore only consider issue reports classified as bugs. Such filtering needs nothing more than a simple database query.

However, all this assumes that the category of the issue report is accurate. In 2008, Antoniol et al. [8] raised the problem of *misclassified* issue reports—reports classified as *bugs*, but actually referring to *non-bug issues*. If such mix-ups (which mostly stem from issue reporters and developers interpreting "bug" differently) occurred frequently and systematically they would introduce *bias* in data mining models threatening the external validity of any study that builds on such data: Predicting the most error-prone files, for instance, may actually yield files most prone to new features. But how often does such misclassification occur and does it actually bias mining models?

These are the questions we address in this chapter. From five open source projects (Section 4.2), we manually classified more than 7,000 issue reports into a fixed set of issue report categories clearly distinguishing the kind of maintenance work required to resolve the task (Section 4.3). Our findings indicate substantial data quality issues:

Issue report classifications are unreliable. In the five bug databases investigated, more than 40% of issue reports are inaccurately classified (Section 4.4)

Every third bug is not a bug. 33.8% of all bug reports do not refer to corrective code maintenance (Section 4.5).

After discussing the possible sources of these misclassifications (Section 4.6), we turn to the consequences. We find that the validity of studies regarding the distribution and prediction of bugs in code is threatened:

Files are wrongly marked as fixed. Due to misclassifications, 39% of files marked as defective actually have never had a bug (Section 4.7).

Files are wrongly marked to be error-prone. Between 16% and 40% of the top 10% most defect-prone files do not belong in this category after reclassification (Section 4.8).

Section 4.9 details studies affected and unaffected by these issues. After discussing threats to validity (Section 4.10), we close with conclusion and consequences (Section 4.11).

Table 4.1: Details of projects used for manual issue report classification.

Project	Maintainer	Tracker type	# Reports
HttpClient	Apache	Jira	746
Jackrabbit	Apache	Jira	2,402
Lucene	Apache	Jira	2,443
Rhino	Mozilla	Bugzilla	1,226
Tomcat5	Apache	Bugzilla	584

4.2 Study Subjects

We conducted our study on five open-source Java projects described in Table 4.1. We aimed to select projects that were under active development and were developed by teams that follow strict commit and bug fixing procedures similar to industry. We also aimed to have a more or less homogeneous data set, which eased the manual inspection phase. Projects from Apache and Mozilla seemed to fit our requirements best. Additionally, we selected the five projects such that we cover at least two different and popular bug tracking systems: Bugzilla[1] and Jira[2]. Three out of five projects (Lucene, Jackrabbit, and HttpClient) use a Jira bug tracker. The remaining two projects (Rhino, Tomcat5) use a Bugzilla tracker.

For each of the five projects, we selected all issue reports that were marked as being *RESOLVED*, *CLOSED*, or *VERIFIED* and whose resolution was set to *FIXED* and performed a manual inspection on these issues. We disregarded issues with resolution in progress or not being accepted, as their features may change in the future.

The number of inspected reports per project can be found in Table 4.1. In total, we obtained 7,401 closed and fixed issue reports. 1,810 of these reports originate from the Rhino and Tomcat5 projects and represent Bugzilla issue reports. The remaining of the 5,591 reports were filed in a Jira bug tracker.

[1]http://www.bugzilla.org/
[2]http://www.atlassian.com/JIRA

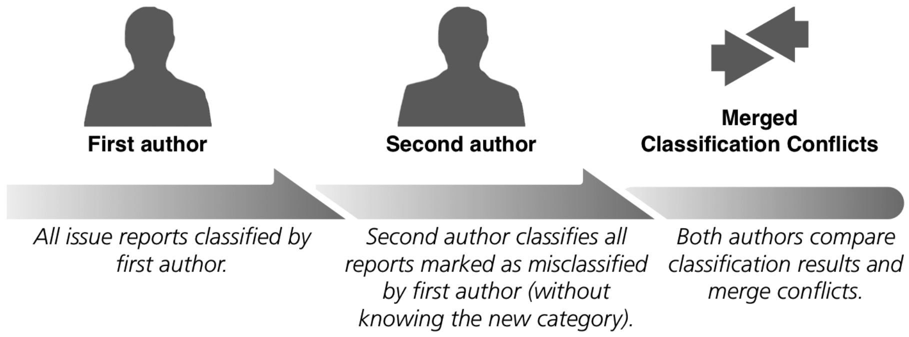

First author
Second author
Merged Classification Conflicts
All issue reports classified by first author.
Second author classifies all reports marked as misclassified by first author (without knowing the new category).
Both authors compare classification results and merge conflicts.

Figure 4.1: The manual report inspection process.

4.3 Manually Classifying Bug Reports

To validate the issue categories contained in the project bug databases, we manually inspected all 7,401 issue reports and checked if the type of each report reflects the maintenance task the developer had to perform in order to fix the corresponding issue. For our manual inspections, we used (a) the issue report itself, (b) all the attached comments and discussions, as well as (c) the code change that was applied to the source code. We analyzed code changes if and only if neither the issue report nor its comments clarified the underlying problem of the reported issue. Each issue report was then categorized into one of eleven different issue report categories shown in Table 4.2.

To assign issue reports to one of the categories, we used a *fixed set of rules* that describe how to classify issue reports based on specific issue report properties. If none of these rules applied, and if we were not able to understand the underlying problem even when inspecting a possible attached patch, we left the original category unchanged. Hence, we favored possible original misclassification noise over new misclassification noise introduced by manual misclassification. The rule set used for classification is given in Section 4.3.1. For each category, we also present a typical real world example.

Table 4.2: The issue report categories used for manual classification.

Category	Description
BUG	Issue reports documenting corrective maintenance tasks that require semantic changes to source code.
RFE	Issue reports documenting an adaptive maintenance task whose resolving patch(es) implemented new functionality (request for enhancement; feature request).
IMPR	Issue reports documenting a perfective maintenance task whose resolution improved the overall handling or performance of existing functionality.
DOC	Issue reports solved by updating external (e.g. website) or code documentation (e.g. JavaDoc).
REFAC	Issues reports resolved by refactoring source code. Typically, these reports were filed by developers.
OTHER	Any issue report that did not fit into any of the other categories. This includes: reports requesting a backport (**BACKPORT**), code cleanups (**CLEANUP**), changes to specification (rather than documentation or code; **SPEC**), general development tasks (**TASK**), and issues regarding test cases (**TEST**). These subcategories are found in the public dataset accompanying this paper.

The manual classification was conducted in three phases as shown in Figure 4.1:

1. In the first phase, the first author inspected all 7,401 issue reports and assigned a report category using the set of report classification rules.
2. The second author re-classified the set of issue reports that were considered to be misclassified after phase one. Again, the second author was using the fixed set of classification rules and the issue reports only; he had no access to the classification results of the first phase. Overall, 3,093 misclassification candidates got reinspected.
3. We then compared the classification results from phase one and phase two to detect classification conflicts—issue reports that were classified differently by the first and the second author. This affected 340 of the 3,093 re-inspected issue reports; the other 94% were independently classified identically by the first and second author and thus validated the accuracy and complexness of the rule set. Each classification conflict finally got resolved by a joint pair-inspection of both authors, partially inducing clarification and refinements of the rule set. (Section 4.3.1 lists the final rule set.)

The first two phases of the inspection process were processed by one individual each. This ensures that all issue reports across all projects are treated and categorized equally. Every issue report reported as misclassified in this paper was independently verified. We did not double check whether the first author did oversee misclassified reports. This implies that the presented misclassification ratios and impact measurements can be considered as a lower bound. The effort for the 10,884 inspections was 4 minutes per issue report on average, totaling 725 hours, or 90 working days.

4.3.1 Classification Rules

A report is categorized as BUG (Fix Request) if...

1. it reports a *NullpointerException* (*NPE*).
2. the discussion concludes that a semantic code change was applied to perform a corrective maintenance task.
3. it fixes runtime or memory issues cause by defects such as endless loops.

Example: TOMCAT5 *report 28147*[3] *is categorized as* **RFE** *but reports a bug that causes a "JasperException for jsp files that are symbolic links". The underlying issue was that tomcat used canonical instead of absolute paths. The applied fix touches one line replacing one method invocation. According to Rule 2, we classified the applied code change as a corrective maintenance task and thus the issue report as* **BUG***.*

A report is categorized as RFE (Feature Request) if...

1. it requests to implement a new access/getter method.
2. it requests to add new functionality.
3. it requests to support new object types, specifications, or standards.

Example: LUCENE *report LUCENE-2074*[4] *is categorized as* **BUG***. But the applied patch and the discussion unveil that a new versioning mechanism had to be implemented. The first comment by Uwe Schindler makes it explicit: "Here the patch. It uses an interface containing the needed methods to easyliy [sic] switch between both impl. The old one was deprecated [...]". This is reclassified as* **RFE** *by Rule 2.*

A report is categorized as IMPR (Improvement Request) if...

1. it discusses resource issues (time, memory) caused by non-optimal algorithms or garbage collection strategies.
2. it discusses semantics-preserving changes (typos, formatting) to code, log messages, exception messages, or property fields.
3. it requests more or fewer log messages.
4. it requests changing the content of log messages.
5. it requests changing the type and/or the message of Exceptions to be thrown.
6. it requests changes supporting new input or output formats (e.g. for backward compatibility or user satisfaction).
7. it introduces concurrent versions of already existent functionalities.

[3] https://issues.apache.org/bugzilla/show_bug.cgi?id=28147
[4] https://issues.apache.org/jira/browse/LUCENE-2074

8. it suggests upgrading or patching third party libraries to overcome issues caused by third party libraries.

9. it requests changes that correct/synchronize an already implemented feature according to specification/documentation.

Example: JACKRABBIT *report JCR-2892*[5] *is filed as* ***BUG*** *under the title "Large fetch sizes have potentially deleterious effects on VM memory requirements when using Oracle". The algorithm fetches data from a database with a large amount of columns and rows, which caused the Oracle driver to allocate a large buffer. The resolution was to develop a new algorithm consuming less memory. This is an* ***IMPR*** *according to Rule 1 since no new functionality was implemented and since the program did not contain any defect.*

A report is categorized as DOC (Documentation Request) if...

1. its discussion unveils that the report was filed due to missing, ambiguous, or outdated documentation.

Example: TOMCAT5 *bug report 30048*[6] *fixes the problem"Setting compressableMimeTypes is ignored." by "Docs updated in CVS to reflect correct spelling." This is a* ***DOC***.

A report is categorized as REFAC (Refactoring Request) if...

1. it requests to move code into other packages, classes, or methods.

2. it requests to rename variables, methods, classes, packages, or configuration options.

Example: TOMCAT5 *report 28286*[7] *is filed as* ***BUG*** *and contains a patch adding a new interface SSOValve. But in comment 4, Remy Maucherat refuses to apply the patch and the idea to introduce a new interface. Instead, he commits a patch that refactors class AuthenticatorBase to allow subclassing. This is a* ***REFAC*** *as per Rule 2.*

[5]https://issues.apache.org/jira/browse/JCR-2892
[6]https://issues.apache.org/bugzilla/show_bug.cgi?id=30048
[7]https://issues.apache.org/bugzilla/show_bug.cgi?id=28286

A report is categorized as OTHER if...

1. it reports violations of JAVA contracts without causing failures (e.g. "*equals()* but no *hashCode()*").
2. complains about compatibility fixes (e.g. "should compile with GCJ").
3. the task does not require changing source or documentation (like packaging, configuration, download, etc.)

Example: *LUCENE report LUCENE-1893[8] complains that "classes implement equals() but not hashCode()". This violated JAVA contracts but does not cause failures. LUCENE report LUCENE-289[9] requests "better support gcj compilation". According to our rules this is considered to be an compatibility improvement classified as* ***OTHER***.

4.4 Amount of Data Noise

In this section, we show the amount of data noise and bias (with respect to issue report types) that is evident in the bug databases of the five analyzed projects (see Section 4.2). We start analyzing the issue report data sets by measuring the false positive rates and slicing individual categories to show how many issue reports were misclassified and which categories these misclassified reports belong to. Later, we will discuss the impact and bias rates for data sets that map issue reports to code changes and source files before we measure the impact on models identifying the most defect-prone files.

As overall noise rate we measured the *false positive rate.* The false positive rate represents the ratio between misclassified issue reports and all issue reports in the data set. The noise rate is independent from individual issue report categories. We will discuss individual categories in Section 4.5. The higher the noise rate, the higher the threat that the noise might cause bias in approaches based on these data sets.

RQ9.1 How much noise due to issue report misclassification exists in bug databases?

Table 4.3 shows the noise rate values for all five projects and for a combined data set containing the issue reports of all five projects. The noise rates for all projects

[8]https://issues.apache.org/jira/browse/LUCENE-1893
[9]https://issues.apache.org/jira/browse/LUCENE-289

Table 4.3: Issue report type rates for all projects and for a combined data set.

Project	**Noise rate**
HttpClient	47.8%
Jackrabbit	37.6%
Lucene	46.4%
Rhino	43.2%
Tomcat5	41.4%
All projects combined	42.6%

lie between 37% and 47%. The overall noise rate lies at 42.6%—that is, two out of five issue reports are wrongly typed. This unexpected high ratio raises threats to any approach based on raw issue report data sets.

☛ *Over all five projects researched, we found 42.6% of all issue reports to be wrongly typed.*

The noise rates of the individual report categories and their variances are shown in Figure 4.2 as box plot. We excluded the categories **DOC** and **REFAC** from this plot since none of the analyzed bug tracking systems supported these report types. The boxes representing the categories **IMPR** and **OTHER** are based on the Jira projects only since Bugzilla does not support these types of reports. Although the noise rates for **IMPR** reports show more variance all projects show comparable noise rates. The variance for **RFE** reports is huge and is partially caused by the fact that the overall number of **RFE** reports is low. Most feature requests have their origin from within the project, especially in open source projects, and it is questionable if or how many such feature requests are documented using a bug tracker.

4.5 Bugs vs. Features

We have seen that two out of five issue reports are misclassified. And we have seen that there exist misclassified **BUG** reports. This is a threat for all empirical studies based on raw, unchecked bug data sets. To raise the level of detail, we sliced issue categories to show the percentage of issue reports that were associated with a category but marked as misclassified. We also include the individual categories the reclassified bug reports belong to when using our classification rule sets presented in Section 4.3.1.

RQ9.2 Which percentage of issue reports associated with a category was marked as misclassified? Which category do these misclassified reports actually belong to?

Each slice contains the set of all issue reports originally associated to a given category and shows to which category the individual issue report actually belongs to (Tables 4.4—4.6). Thus, each slice table cell contains the percentage of issue reports originally associated to a given category that were manually classified into the issue category indicated by the row name. The lengths of the bars behind the percentage numbers represent the individual percentage visually. The last row of each slice states the percentage of reports originally associated to the corresponding category that were assigned a different category during manual classification. The values of this last row correspond to the category boxes in the box plot shown in Figure 4.2.

4.5.1 Bugs

Table 4.4 contains the noise rate slice for the **BUG** issue category. We already discussed in Section 4.4 that the noise rate for **BUG** reports is surprisingly high for all projects. Tracking bug reports and their target categories shows that 13% of **BUG** reports are manually classified into the **OTHER** category containing multiple sub-categories (see Section 4.3). Between 6% and 13% of filed **BUG** reports are improvement requests and up to 10% contain documentation issues. The fraction of bug reports containing feature requests lies between 2% and 7%. The striking number, however, is that on average 33.8% of all issue reports are misclassified.

☛ *Every third bug report is no bug report.*

The noise rate slice for bug reports is of great importance. Bug reports are one of the most frequently used instruments to measure code quality when being mapped to code changes. But feature requests, improvement requests, and even documentation issues can also be mapped to code changes implementing a new feature, implementing an improvement, or fixing code comments. Thus, we cannot rely on natural filtering mechanisms that rule out misclassified **BUG** reports belonging to any report category

Table 4.4: Reclassification of reports originally filed as **`BUG`**

Classified category	HttpClient	Jackrabbit	Lucene	Rhino	Tomcat5	Combined
`BUG`	63.5%	75.1%	65.4%	59.2%	61.3%	66.2%
`RFE`	6.6%	1.9%	4.8%	6.0%	3.1%	3.9%
`DOC`	8.7%	1.5%	4.8%	0.0%	10.3%	5.1%
`IMPR`	13.0%	5.9%	7.9%	8.8%	12.0%	9.0%
`REFAC`	1.7%	0.9%	4.3%	10.2%	0.5%	2.8%
`OTHER`	6.4%	14.7%	12.7%	15.8%	12.9%	13.0%
Misclassifications	36.5%	24.9%	34.6%	40.8%	38.7%	33.8%

Table 4.5: Reclassification of reports originally filed as **`RFE`**

Classified category	HttpClient	Jackrabbit	Lucene	Rhino	Tomcat5	Combined
`BUG`	0.0%	0.7%	0.0%	3.6%	8.1%	2.8%
`RFE`	100.0%	91.3%	97.0%	42.9%	39.6%	72.6%
`DOC`	0.0%	2.0%	0.0%	0.0%	18.1%	5.3%
`IMPR`	0.0%	0.7%	0.6%	19.0%	20.8%	8.6%
`REFAC`	0.0%	0.0%	0.0%	15.5%	3.4%	3.2%
`OTHER`	0.0%	5.3%	2.4%	19.0%	10.1%	7.5%
Misclassifications	0.0%	8.6%	3.0%	57.1%	60.4%	24.7%

Table 4.6: Reclassification of reports originally filed as **`IMPR`**.

Classified category	HttpClient	Jackrabbit	Lucene	Rhino	Tomcat5	Combined
`BUG`	2.6%	2.8%	1.8%	0.0%	0.0%	2.3%
`RFE`	45.3%	18.8%	28.6%	0.0%	0.0%	26.1%
`DOC`	11.6%	3.7%	7.2%	0.0%	0.0%	6.2%
`IMPR`	26.7%	45.6%	35.2%	0.0%	0.0%	38.8%
`REFAC`	4.3%	9.2%	14.2%	0.0%	0.0%	10.9%
`OTHER`	9.5%	19.8%	13.0%	0.0%	0.0%	29.4%
Misclassifications	73.3%	54.4%	64.8%	0.0%	0.0%	61.2%

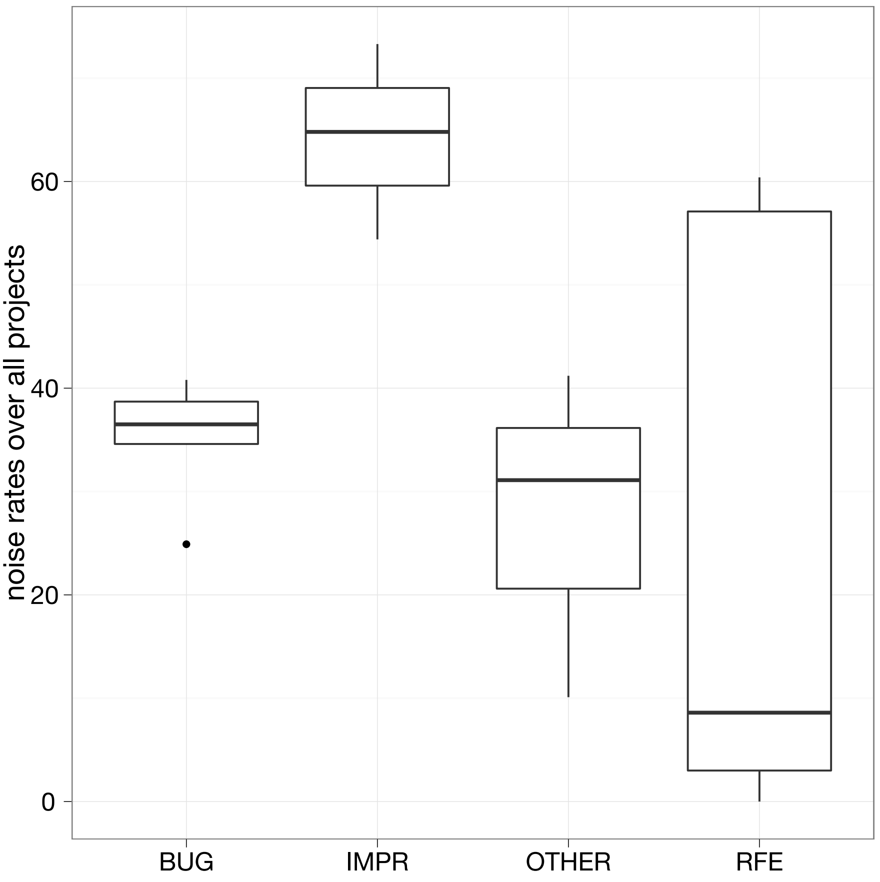


Figure 4.2: The noise rates split by category over all projects. The categories **DOC** and **REFAC** are not present since none of the bug trackers supports these categories. The Bugzilla projects are only included in the analysis of **BUG** and **RFE** reports since Bugzilla does not support any of the other issue report categories.

that will not cause code changes being applied to source files. Studies that use bug data sets might be impacted by data noise as shown in Table 4.4. The noise rates in this section include issue reports that might not be mapped to code changes or files. We will discuss the bias caused by bug data noise later in this paper.

4.5.2 Feature Requests

The noise rate slice for issues originally categorized as **RFE** is interesting because it shows a fundamental difference between BUGZILLA and JIRA trackers. As you can see in Table 4.5 the false positive rates for all three JIRA projects lie between zero and nine percent. The corresponding false positive rates jump to 57% and 60% for BUGZILLA trackers. Interpreting these values, it seems that issue reports in BUGZILLA trackers are less reliable than JIRA reports. This matches the fact that the false positive rates for **BUG** reports in Table 4.4 where larger for BUGZILLA trackers, too. By default, BUGZILLA trackers support less issue report types than JIRA. This has the consequence that reporters and developers that file issue tickets not being bug reports use the only alternative label **RFE**.

4.5.3 Improvement Requests

The last noise rate slice shows how many improvement requests were differently categorized during manual inspection (see Table 4.6). The columns for BUGZILLA tracker projects remain zero since by default BUGZILLA trackers do not support these issue categories. For the remaining three projects, between 19% and 45% of improvement requests were manually categorized as **RFE** issue reports. Only a very marginal low fraction of 2% were manually classified as bug reports. On average, more than 60% of improvement requests were reclassified during manual inspection.

4.6 Sources of Misclassification

The misclassification ratios presented in the last section shed a low light on the data quality of bug databases. But why do issue tracking systems contain so many misclassified reports? For us, the main reason is that *users and developers* have very different views and insights on bug classification, and that *classification is not rectified* once a bug has been resolved.

Bug tracking systems are communication tools that allow users to file bug reports that will be fixed by developers. But users and developers do not share the same perspective regarding the project internals. In many cases, users have no project insight at all and might not even have the ability to understand technical project details. Users tend to consider every problem as a bug. From their perspective, the software does not comply with their expectations or with the provided documentation and so they file a bug report. A user filing an issue report might not even know the difference between improvement, feature request, or bug report. But it is the *reporter* who assigns an issue category.

On the other side, the developer is the expert of any technical detail of the program; she designed and implemented it. This difference between reporter and resolver already is a source of uncertainty. In contrast to a reporter, a developer certainly has the ability to distinguish between different problems and the required maintenance task to solve the issue. The developer would be the right person to categorize issue reports. But this is not how bug trackers work. Of course, the developer could change the issue category after resolution—but this happens rarely. In many cases there exists no real motivation to change the issue category once the cause for a problem is found and fixed.

This conceptional problem may explain many of the misclassification patterns we observed during manual inspection. It also explains the high misclassification noise rates for **BUG** reports. Using their default configuration, many bug tracking systems set the report type to **BUG** by default. Combining this technical limitation with the above discussed problem that the potentially more inexperienced communication partner decides which report type to be assigned, we are left with many **BUG** reports that should have been filed as improvement request or even feature request.

The question is whether these misclassification sources impact issue reports that can actually be mapped to source code changes and thus to source files. Consider a user filing a bug report complaining about a documentation issue. To resolve this issue, the developer might have to change the code documentation contained in the source file. So we would map the report into source code and count it as a bug fix although the plain source code did not change. And this is why **DOC** issues originally filed as **BUG** are dangerous. We cannot rely on automatic filters that rule out any report that did not change any source file.

4.6.1 Ambiguous Terms

Many misclassifications stem from ambiguous terms and vague definitions. In this paper we showed that BUGZILLA contains many **RFE** reports that do not contain feature requests (see Section 4.4). The main source for this misclassification lies in BUGZILLA itself. In default configuration, BUGZILLA offers only two different issue categories: **BUG** and *Enhancement*. But the term of an enhancement is ambiguous. Fixing a bug can be seen as an enhancement but we as data miners would like to see bug fixes being classified as **BUG**. During manual classification we considered enhancements as feature requests. And indeed, the largest group of the enhancement requests are requests for implementing new features although the majority of enhancement requests should not be marked as **RFE** (see Table 4.5). You could also mark BUGZILLA enhancements as **IMPR** but you have to make a decision when mining and no matter how you decide, you will end up with issue misclassification noise.

Similar reasons exist for the high proportion of JIRA improvement requests being manually classified to **RFE**. Here we run into a general definition problem: "When does a code change apply a feature implementation and when does it apply an improvement?" Since we classify issue reports from a developer perspective, we considered code changes adding any new functionality (e.g. adding a getter method or adding new API methods) as **RFE**. In the context of this paper, improvements are strictly bound to maintenance tasks improving existing code fragments and algorithms. Across all projects, there exists a number of reports requesting documentation improvements ("The error message contains a misleading error cause"), build improvements ("The project does not build on architecture X"), or test improvements ("Running the test suite takes too long"). In line with bug reports targeting non-functional properties of the source code, we classified these improvement requests as **DOC** or **OTHER** issues.

4.7 Impact on Mapping

The issue report misclassification noise presented in Sections 4.4 and 4.5 can impact studies and tools that use these or similar data sets without validating them. As a first category, we discuss how misclassified issue reports impact approaches that map issue reports to source code changes—for instance, to identify files that had the most bugs in the past.

RQ9.3 What is the impact of misclassified issue reports when mapping issue reports to source code changes?

For this purpose, we followed the issue report mapping strategy described by Zimmermann et al. [153], a mapping method frequently replicated by many studies. Scanning through the commit messages contained in a version archive, we detect issue report identifiers using regular expressions and key words. Once we mapped issue reports to version archive revision, we can identify the set of issue reports that caused a change within the source file. Ignoring report severity we then count the number of distinct issue reports originally classified as **BUG** (*num_original_bugs*) for each source file of a given software project. Additionally, we count the number of distinct issue reports that were classified as **BUG** during manual inspection (*num_classified_bugs*). We measure the issue mapping bias using five different bias measurements.

MappingBiasRate: This bias rate expresses the percentage of false positive original **BUG** reports that could be mapped to code files. The *mappingBiasRate* corresponds to the false positive rates shown in Figure 4.2 but is limited to **BUG** reports that can be mapped to code changes.

DiffBugNumRate: The *diffBugNumRate* represents the number of files for which

$$num\_original\_bugs - num\_classified\_bugs \neq 0.$$

The measure ignores files for which the set but not the number of issue reports differ. Counting the fixes does not require the individual report to be known.

MissDefectRate: The *missDefectRate* is defined as

$$missDefectRate = \frac{numMissDefect}{numZeroOriginalDefect}$$

where *numMissDefect* represents the number of source files for which no original bug report could be mapped but that have at least one manually classified bug report assigned and where *numZeroOriginalDefect* is the number of source files having no original bug report assigned. This measure is important for defect classification models (distinction between has bug or has no bug).

FalseDefectRate: Analog to *missDefectRate*, we compute the *falseDefectRate* as

$$falseDefectRate = \frac{numFalseDefect}{numOriginalDefect}$$

where *numFalseDefect* is the number of source files with at least one original bug report assigned but no manually classified bug report. *numOriginalDefect* equals the total number of source files that got at least one original bug report assigned.

Table 4.7: Impact of misclassified issue reports on mapping strategies and approaches.

Measure	HttpClient	Jackrabbit	Lucene	Rhino	Tomcat5	Average
MappingBiasRate (False positive rate for mappable **BUG** reports)	24%	36%	21%	38%	28%	29%
DiffBugNumRate (How many files will change their defect-prone ranking?)	62%	17%	14%	52%	39%	37%
MissDefectRate (How many files with no original **BUG** have at least one classified **BUG**?)	1%	0.3%	0.7%	0%	38%	8%
FalseDefectRate (How many files with at least one original **BUG** have no classified **BUG**?)	70%	43%	29%	32%	21%	39%

The values of these bias measures for our five target projects are shown in Table 4.7 along with an additional column containing the average bias measures. For all projects, the number of misclassified **BUG** reports that can be mapped to source files (*mappingBiasRate*) lies above 20%. On average, every third mappable bug report is misclassified. This is a threatening high fraction and confirms that the misclassification noise rates presented in Section 4.4 also affect issue reports that can be mapped to source code changes. On average, the *mappingBiasRate* is only five percent points below the average false positive rate for bug reports shown in Table 4.4. The *mappingBiasRate* indicates that bug tracking systems and their different usage behavior seem to have no impact on the mapping bias.

The second row of Table 4.7 shows the fraction of files having a different number of mapped bug reports. The *diffBugNumRate* shows how many files will change their defect-prone ranking. This value might also have severe consequences for defect prediction models based on concrete bug count numbers (see Section 4.8). On average 37% of all source files have biased bug count numbers. For the projects HttpClient and Rhino the *diffBugNumRate* well exceeds the 50% margin.

Row three and row four of Table 4.7 are interesting for approaches using classification models grouping source files into two groups of defect-prone and non defect-prone entities. The fractions of files that were falsely marked as defect free (*missDefectRate*) is very low and can be disregarded, except for Tomcat5. But the fraction of false classified defect-prone using a threshold of one to distinguish between defect-prone and non defect-prone entities (*falseDefectRate*) is significant. 20% to 70% of the original defect-prone marked source files contained no defect. An average *falseDefectRate* of 39% shows that mapping bias is a real threat to any defect prone classification model.

☛ *On average, 39% of all files marked as defective actually never had a bug.*

To give some more details on the differences between original and classified bug counts (*diffBugNumRate*), Figure 4.3 shows stacked bars displaying the distribution of bug count differences among source files. Each stacked bar contains intervals reflecting the difference between *num_original_bugs* and *num_classified_bugs*. A positive difference indicates that the number of defects fixed in the corresponding source files is actually lower. For files showing a negative difference more defect fixes could have been found. While most files show no or only little changes to their bug count there also exist files for that between five and 26 bugs were wrongly counted. The number of files for which more bugs could have been found is marginal.

4.8 Impact on Bug Prediction

The results presented in the last section indicate that defect prediction models based on bug data sets containing misclassified bug reports might be severely biased. To verify this threat, we conducted an experiment that uses a simple quality model that identifies the most defect-prone source files by counting the number of distinct bug reports mapped to the corresponding file. If we can show that such a simple bug count model is affected, more complex models based on similar count or classification schemata will be affected too.

RQ9.4 How does bug mapping bias introduced by misclassified issue reports impact the TOP 5%, 10%, 15%, 20% of most defect prone source files?

The experiment to answer RQ9.4) is visually described in Figure 4.4. We duplicate the set of source files and sort each copy by two different criteria. One set gets sorted in descending order using the number of original bug reports (*num_original_bugs*). The other set clone gets sorted in descending order using the number of manually classified bug reports (*num_classified_bugs*). In each set, the most defect-prone file is the top element. Comparing the top X% of both file sets (containing the same elements but in potentially different order) allows us to reason about the impact of mapping bias on models using bug counts to identify the most defect-prone entities. Since both cutoffs are equally large (the number of source files does not change, only their ranks), we can define the *cutoff_difference* as:

$$\frac{\text{size of cutoff} - \text{size of intersection}}{\text{size of cutoff}}.$$

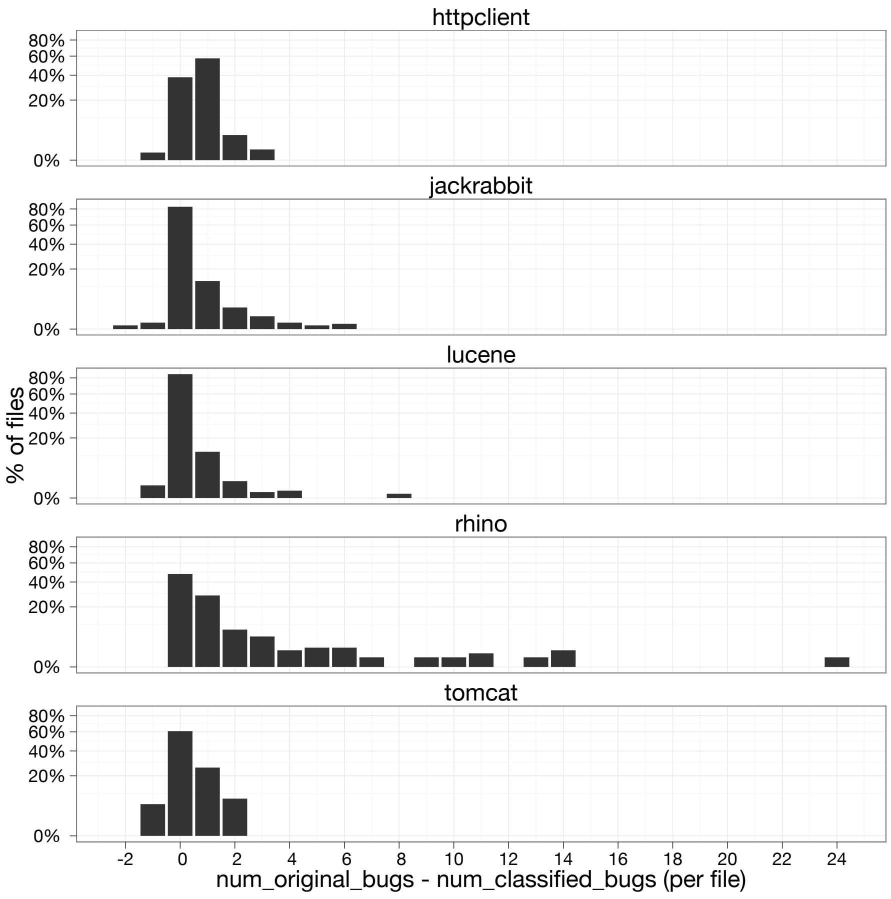


Figure 4.3: Histograms showing *diffBugNumRates* and their frequencies across all five projects. For files with a *diffBugNumRate* of zero the number of associated bugs remained equal. Files with a positive diff rate had too many bug reports assigned.

The result is a number between zero and one where zero indicates that both cutoffs are identical and where a value of one implies two cutoffs with an empty intersection. A low cutoff_difference is desirable.

Figure 4.5 contains the *cutoff_differences* for all five projects using the top 5%, 10%, 15%, and 20%. The *cutoff_differences* is stable across projects and cutoff sizes. Considering the top 5% cutoff the *cutoff_differences* lie between 11% and 29% and raise to a range between 16% and 35% for a cutoff size of 20%. The variance between the different cutoff sizes per project lies around 15% for HttpClient and Tomcat5 and 5% for Lucene and Rhino. But the bias measured for all projects and cutoffs lies well above 10%. The comparable relative stable results across all projects and cutoff sized show that quality measuring approaches using biased report to code mappings would report a false positive rate between 16% and 40% for the top 10% most defect-prone files.

☛ *When predicting the top 10% most defect-prone files, 16% to 40% of the files do not belong in this category because of misclassification.*

Table 4.8 and Figure 4.6 show the Spearman rank correlations for all source files in the corresponding intersections. These rank correlations indicate the relative order for those files that remain in the top X% most defect-prone source files. A correlation value of one means that there exist files that should not belong to the top cutoff but at least the relative order of the correctly classified files remains stable. All rank correlation for HttpClient are above 0.7 suggesting that the relative order or correctly classified files is least impacted. Except for Rhino top 10% and Tomcat5 top 5%, the rank correlations lie below 0.7 and reach correlation values close to zero and even below zero.

☛ *Misclassification also impacts the relative order of the most defect-prone files.*

4.9 Implications on Earlier Studies

The results presented in this chapter show that misclassified issue reports can affect the assessment and prediction of code quality based on bug data sets. Hence, empirical studies that use or used bug data sets without validating them might suffer from bias.

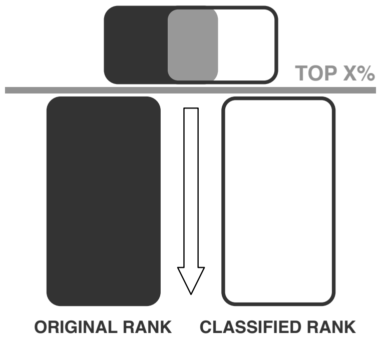


Figure 4.4: The cutoff_difference for the top x% illustrating the impact of misclassified issue reports on quality data models.

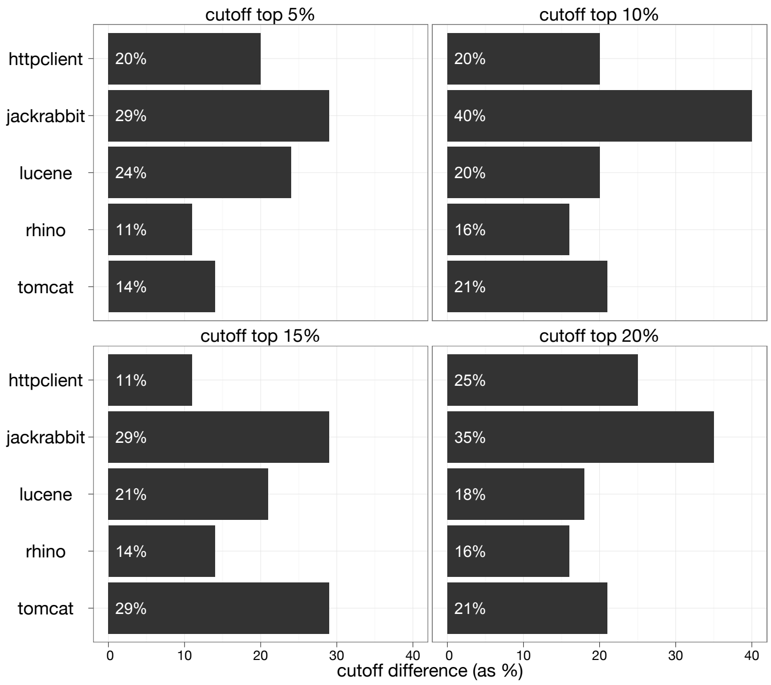


Figure 4.5: The cutoff_differences caused by misclassified bug reports.

Table 4.8: Spearman rank correlations for all source files remaining in the intersection of original and classified most defect-prone entities.

	TOP 5%	**TOP 10%**	**TOP 15%**	**TOP 20%**
HttpClient	1	0.8	0.8	0.7
Jackrabbit	0.2	0.3	0.5	0.6
Lucene	0.4	0.5	0.4	0.2
Rhino	−0.1	0.7	0.4	0.4
Tomcat5	0.8	0.5	0.5	0.6

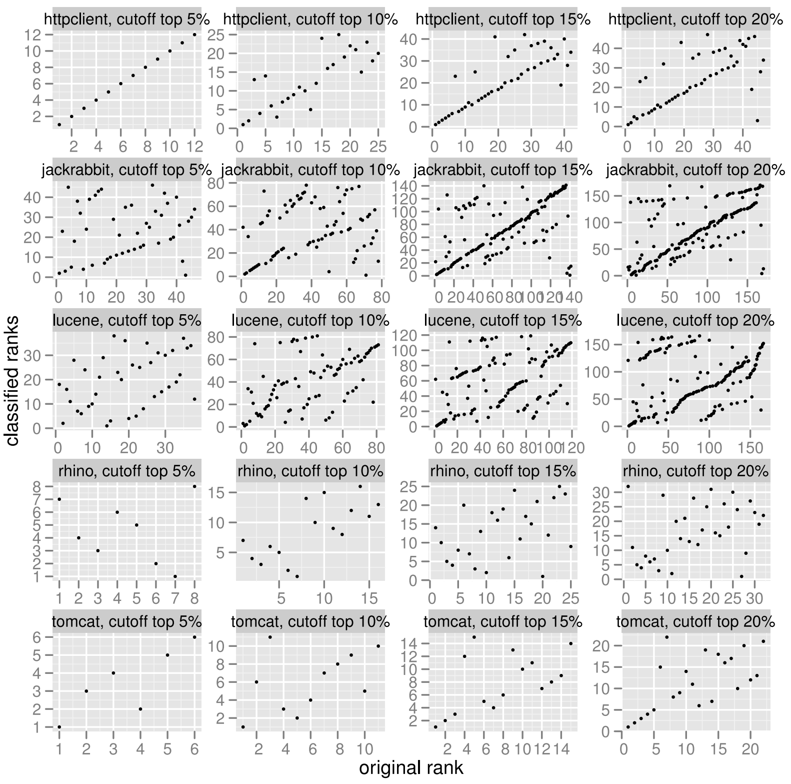


Figure 4.6: Spearman rank correlations for all source files remaining in the intersection of original and classified most defect prone entities.

The approaches and their evaluations remain valid—training and testing machine learners on equally biased training and testing sets does not harm the models themselves. But biased data sets raise questions whether the trained and tested models actually fit the desired goal. As an example, if half of the issue reports treated as **BUG** reports are actually **RFE** reports, the resulting "defect" prediction models are likely to predict code changes in general but not bug fixes. Even if there exist a high correlation between the number code changes and bug fixes per source artifact, the models trained and tested on the biased data sets do not deliver the answers to a given research question.

The list of threatened studies and approaches mentioned in this section is neither representative nor complete. Our aim is to give a brief overview over studies that are likely (but not proven) to be biased. Any study reusing these experimental setups without performing extra bug data validation is likely to be biased as well.

4.9.1 Studies threatened to be biased

Mapping bugs to code changes was first introduced by Fischer et al. [45] and Čubranić et al. [137] who described procedures to created a release history database from version control and bug tracking systems and to map bug reports to code changes. These two approaches do not interpret the mapped artifacts and are per se not threatened; however, any study using one of these approaches to derived code quality measures is likely to be threatened if it did not perform additional data validation.

The list of papers affected encompasses more than 150 published studies in the ACM digital library citing these two approaches (as of August 2012). Zimmermann et al. [153] is a particular important case, as a large number of papers built on the accompanying (now found to be biased) bug data set. Taking the blame, other typical examples with one of us as co-authors are: [89, 110, 119, 126, 129].

The threat to validity for all these papers is that the bug data set they have been evaluated on contains a mix of bugs and non-bugs. Hence, in their evaluation, they map and predict non-bugs as well as bugs. Users would be generally interested in predicting bugs rather than non-bugs, however; and we now no longer know how these approaches perform and compare when using a data set consisting only of true bugs. This threatens their external validity.

Construct and internal validity remain unquestioned, though: the approaches and techniques are still original and valid, and can still provide good results. It may even be that filtering out non-bugs yields less noise in terms of predictor features, and thus generally improves results. Assessing such effects for all earlier studies is beyond the scope of this paper; however, we provide rectified data for future studies.

How about studies using industrial bug data? Since we have no insights on the data quality of industrial data sets, these studies may or may not be threatened. In general, one might hypothesize that industry has stronger process rules and incentives (speak: measurements and goal metrics), which encourage accurate issue classification. Lately, Nguyen et al. provided evidence that "even in such a perfect setting, with a near-ideal dataset, bias do exist [...]" [111]. Still, a more strict development process and thus less bias in corresponding data sets could explain why predictors such as change bursts [110] or network metrics [151] work extremely well on industrial data sets, but poorly on open source data sets.

Exploring the quality of such data sets and differences between industrial and open source projects again is a topic for future research.

4.9.2 Preventing misclassification threats

How can we improve the quality of bug datasets? The following approaches all help:

Test cases. Approaches like *iBugs* [39] validate bug reports using test cases to replicate bugs. This straightforward filtering mechanism ensures each bug is valid. Consequently, studies relying on the *iBugs* data sets are not affected by issues discussed in this paper.

Code history. Kim et al. [86] uses version control history to verify that applied code changes are actual fixes. This approach solely relies on code evolution and thus is not sensitive to bug database issues. Again, this is a recommended procedure to prevent misclassification.

Automatic classification. Automatic classification models as described by Antoniol et al. [8] can be used to categorize issue reports based on the text of the issue report itself with precision rates between 77% and 82%. Although, constructing classification training sets requires initial human effort, such predictors should quickly reduce the required human interaction.

Rectified Data Sets. Our data sets rectified by manual bug classification are publicly available (Section 4.11); we encourage their use for further research.

We strongly recommend to use additional data (e.g. tests) or human effort to reduce the high amount of misclassified issue reports. This means that the contributions of the mining software archive field can still all be applied; one just needs a bit of validation in the first place.

4.10 Threats to Validity

Empirical studies like this one have their own threats to validity. We identified three noteworthy threats:

Manual inspection. First and most noteworthy, the manual inspection phase is crucial. To counter the threat of us making classification mistakes, we chose a setup that ensures that every misclassified bug report is cross-validated and that classification conflicts have to pass a third inspection. Still, we cannot rule out that the manual inspection contains errors. Additionally, we make our entire dataset available for independent assessment.

Classification rules. Second, the set of classification rules is only one possibility to classify issue reports. There exists no clear definition separating feature and improvement requests. Using a different classification rule set will certainly impact the results presented in this paper. We counter this threat by listing the complete rules verbatim.

Study subjects. Third, the projects and bug tracking systems investigated might not be representative, threatening the external validity of our findings. Although JIRA and BUGZILLA are popular bug tracking systems, we cannot ensure that other projects using the same or even other bug tracking systems contain comparable amount and distribution of misclassified issue reports.

4.11 Summary

Mining software archives has long been seen as the full automation of empirical software engineering—all one needs to do is to point the mining tool at a new data source, and out pop the correlations and recommendations. The findings in this paper suggest widespread issues with the separation of bugs and non-bugs in software archives, which can severely impact the accuracy of any tool and study that leverages such data. The consequences are straightforward:

- First and foremost, automated quantitative analysis should always include human qualitative analysis of the input data—and of the findings. Approaches relying on bug datasets should be preceded by a careful manual validation of data quality; at least of a significant sample. Data quality should be discussed as a threat to validity.
- Bug prediction models trained and evaluated on biased data sets are threatened to predict changes instead of bugs. Filtering out non-bugs when estimating code quality might even improve results.
- The categorization of bug reports is dependent on the perspective of the observer. Approaches using bug data sets should be aware of this fact and validate whether the perspective of the prediction model matches the perspective of the bug creator.

Generally, one should always be aware that not all bugs should be treated equal. Many bugs are of little to no consequence, while a few ones–such as security or privacy issues—can easily damage the reputation of the entire product or even threaten the existence of the company. Assessing such consequences cannot be left to machines alone.

Hence, dealing with bug databases will always require human effort—an investment which, however, pays off in the end. Our motivation for this work was to have a well-classified set of bug reports and features, which we now can leverage (and share) for future research. In the long run, better data will lead to better recommendations, and better recommendations in turn will make developers more conscious of maintaining data quality—a virtuous circle in which processes and their metrics can improve in unison.

Detailed references, all data sets (original and rectified), all slices and more information can be found at:

`http://www.softevo.org/bugclassify`

Chapter 5

Untangling Changes

Parts of the contents of this chapter have been published in the technical report Herzig and Zeller [72] (currently under submission).

5.1 Introduction

A large fraction of recent work in empirical software engineering is based on *mining version archives*—analyzing which changes were made to a system, by whom, when, and where. Such mined information can be used to predict related changes [154], to predict future defects [98, 153], to analyze who should be assigned a particular task [9, 18], or simply to gain insights about specific projects [93].

All these studies, including the work presented in this thesis, depend on the *accuracy* of the mined information —accuracy that is threatened by noise. Such noise can come from missing associations between change and bug databases [20] or misclassified issue reports (see Chapter 4). One significant source of noise so far overseen, however, are *tangled change sets*. Section 3.6 contains a discussion on why tangled change sets are a major threat for change genealogies.

What is a tangled change set? Let us assume we have a developer who is assigned multiple tasks *A*, *B*, and *C*. Let all these have a separate purpose; *A* may be a bug

Table 5.1: Proportion of issue-fixing changes that address more than one issue.

	# fixes	**# tangled fixes**
ArgoUML	2,945	171 (5.8%)
Google Webtool Kit	809	68 (8.4%)
Jaxen	105	12 (11.4%)
JRuby	2,977	275 (9.2%)
Xstream	312	37 (11.9%)

fix, *B* may be an extension, and *C* may be a refactoring. When the developer is done with the three tasks, she has to commit her changes to the version control system, such that the changes get propagated to other developers and can go into production. When committing her changes, she may be disciplined and group her changes into three individual commits, each containing the changes pertaining to each task and coming with an individual description. This separation is complicated, though; for instance, the tasks may require changes in similar locations. Therefore, it is more likely that she will commit all changes tangled in a single transaction, with a message such as “Fixed bug #334 in `foo.c` and `bar.c`; new feature #776 in `bar.c`; `qux.c` refactored; general typo fixes”.

Such tangled change sets do not cause serious trouble in development. However, they introduce *noise* in any analysis of the version archive, thereby compromising the accuracy of the analysis. As the tangled change set fixed a bug, all files touched by it will now be marked as having had a defect, even though the tangled tasks *B* and *C* have nothing to do with a defect. Likewise, all files will be marked as being changed together, which may now induce a recommender to suggest changes to `qux.c` whenever `foo.c` is changed. Commit messages such as “general typo fixes” point to additional minor changes all over the code—locations which will now be related with each other as well as the tasks *A*, *B*, and *C*.

The problem of tangled change sets is not a theoretical one. In an exploratory study on five open source projects, we manually classified more than 7,000 individual issue-fixing changes and checked whether these changes addressed multiple (“tangled”) developer tasks. Table 5.1 summarizes our results: *Between 6% and 15% of all fixes address multiple concerns at once*—they are tangled and therefore introduce noise and bias into any analysis of the respective change history (Section 5.2 has more on this study.). These findings confirm results of earlier research studies [83, 84].

The are two main approaches to overcome the problem of tangled change sets. One solution is to detect tangled change sets and to ignore these data points in any further analyses. But this solution makes two major assumptions. First, one must be able to detect tangled change sets automatically; second, the fraction of tangled change sets must be small enough such that deleting these data points does not cause the overall data set to be compromised. The second approach to deal with such tangled change sets is to *untangle* them into separate changes that can be individually analyzed, thereby reducing the noise.

In this chapter, we present an approach to untangle changes. It splits tangled change sets into smaller *partitions*, where each partition contains a subset of change operations that are related to each other, but not related to the change operations in other partitions. The algorithm is based on static code analysis only and is fully automatic, allowing archive miners to untangle tangled change sets and to use the created change partitions instead of the original tangled change set. Our experiments on five open-source projects show that neither data dependencies, distance measures, change couplings, or distances in call graphs serve as a one-size-fits-all solution. By combining these measures, however, we obtain an effective approach that untangles multiple combined changes with a mean precision of 58%–80%.

In his Master thesis, Kawrykow [83] presented a similar approach to detect subtasks in change sets. His approach focuses on connections between updated elements. Our approach developed in parallel but independently shares the same basic approach but operates on a higher level of granularity.

The remainder of this chapter is organized as follows. To motivate untangling, we discuss the causes and effects of tangled change sets in Section 5.2. Then, we discuss the basic principles of our untangling approach, before describing the evaluation setup in Section 5.5. The evaluation results are presented in Section 5.6 and the threats to validity in (Section 5.8). Section 5.9 closes with conclusion and consequences.

5.2 Tangled Changes

Many approaches to mining version archives require quality data used to train machine learning models. Such quality data can be extracted when combining bug databases and version archives. Zimmermann et al. [153] describe one standard approach to generate their historic quality data sets mapping bug fixes to code artifacts. The idea is as follows: one requires a list of *bug fixes* applied to individual code artifacts (e.g.

code files) during project history: The more bugs were fixed over time, the worse the code artifact's original quality. To retrieve such a bug fix count, one identifies bug fixing change sets by parsing commit messages. Once the change set is confirmed to reference a closed and fixed bug report, all code artifacts changed within the change set are marked as being fixed. But such an approach relies on the basic assumption that change sets are *atomic*—each change set contains only those changes that are necessary to fix the referenced issue. However, as discussed in Section 3.1, there may be various reasons to tangle multiple unrelated tasks into a single commit.

When analyzing such tangled change sets, it is not easy to determine which code artifacts were changed due to which developer maintenance task.Instead of mapping developer maintenance tasks to all changed code artifacts within the original change set, one would assign individual developer maintenance tasks to those code artifacts changed within the change set partition. Assigning developer tasks to change set partitions would require change classification techniques automatically deciding whether a code change fixes a bug or implements a new feature. Mockus and Votta [101] showed that bug fixes show significant different impact on code complexity compared to other code changes. Using a automated change classification model (see Chapter 6) we can distinguish between bug fixing and feature implementing change sets. Untangling change sets containing multiple bug fixes —e.g. in bug fixing branches—requires no such classification techniques. To the best of our knowledge, however, all approaches to mining version archives consider change sets as atomic. When it comes to defect prediction, this implies that a bug report is mapped to every single change in the associated change sets; consequently, this will introduce bias into the data set, which may spoil the results of any analysis based on this data.

The bias caused by tangled change sets is significant. Table 5.1 shows the number of change sets applying a fix for at least one issue report (may also be a feature request) for five open-source projects. We considered only those change sets that contained the keywords *fixed*, *resolved*, or *issue* and later manually validated that their commit message marks an issue within the source code (bug and feature requests). We also manually checked the commit messages of these fixes and marked a change set as tangled if the commit message clearly indicated that the applied changes tackle more than one developer maintenance task. This can either be commit messages that contain more than one issue report reference (e.g. "XSTR-93, XSTR-120, XSTR-170: Support for \r newline in strings.") or a commit message indicating extra work committed along the issue fix (e.g."Fixes issue #591[...]. Also contains some formatting and cleanup.")—mostly cleanups and refactorings. The fraction of tangled fixes lies between 6% and 12% (Table 5.1) for issue fixing change sets.

These findings are supported by other studies reporting similar bias figures for change sets. Dallmeier [38] used delta debugging to minimize bug fixes of two open source projects to a minimal set of code changes letting a regression test pass. The results show that on average 50% to 60% of the code changes applied within a bug fix transaction had no effect on the result of the regression tests. Similarly, in a study of over 24,000 change sets from seven open-source projects [84], Kawrykow and Robillard found that 2% to 15% of all method updates were due to *non-essential differences*—code changes that did not change the semantics of the program. Combining the bias fractions caused by non-essential changes and tangled change sets, *up to 30% of all changes are falsely associated with bug reports; they either are tangled with other changes by coincidence, or have no impact on the semantics at all.* The effects of such biased data mining sets are significant [20, 84].

☛ *Up to 30% of all changes are falsely associated with bug reports.*

5.3 The Untangling Algorithm

Many change sets contain change operations serving multiple developer maintenance tasks. Files touched by a single tangled change set may contain changes applied to resolve different developer maintenance tasks and should be separated when analyzing them or mapping developer maintenance tasks to changed files. Otherwise, such tangled change sets lead to harmful bias impacting prediction models based on version archive training sets. To reduce this bias, it is necessary to *untangle* change sets into individual, corresponding *change set partitions*.

Generally, determining unrelated code changes applied together is undecidable, as the halting problem prevents predicting whether a given code change has an effect on a given problem. Consequently, every untangling algorithm will have to rely on heuristics presenting an approximation of how to separate two or more code changes. We cannot solve the untangling problem completely, but we hope to reduce the amount of bias significantly.

An *untangling algorithm* should be fully automatic and simple at the same time. The algorithm proposed in this chapter expects an arbitrary change set as input and returns a set of change set partitions. Each change set partition contains change operations that are related—ideally all change operations necessary to resolve one developer maintenance task (e.g. fixing a bug). The union of all change set partitions equals the original change set.

5.3.1 Related changes

For each pair of applied change operations, the algorithm has to decide whether both change operations belong to the same change set partition (are related) or should be assigned to separate change set partitions (are not related). To determine whether two change operations are related or not, we have to determine the *relation distance* between two code changes such that the distance between two related change operations is significantly lower than the distance between two unrelated change operations. Analyzing the change operations themselves and considering the project's history, there are multiple ways to define distance between two change operations. However, none of them seem to be powerful enough to capture the complexity of change operation relations. For example, it seems reasonable that two change operations changing statements reading/writing the same local variable are very likely to belong together. But vice versa, two code changes not reading/writing the same local variable may very well belong together—if both change operations affect consecutive lines. As a consequence, our untangling algorithm is based on a feature vector spanning multiple aspects describing the distances between individual change operations and should combine these distance measures to separate related from unrelated change operations.

In Section 5.4, we will discuss how to combine multiple distance measures to decide which code changes are likely to be related. But before that, let us discuss how the overall algorithm is designed.

5.3.2 Using Multilevel Graph Partitioning

From the previous section we learned that the untangling algorithm has to iterate over pairs of change operations and needs to determine the likelihood that these two change operations are related and thus should belong to the same change set partition. Although we do not partition graphs, we reuse the basic concepts of a general multilevel graph-partitioning algorithm proposed by Karypis and Kumar [80, 81, 82]. We use a triangle partition matrix to represent existing untangling partitions and the confidence values indicating how confident we are that two corresponding partitions belong together. We will start with the finest granular partitioning and merge those partitions with the highest merge confidence value. After each partition merge we delete two partitions and add one new partition representing the partition union of the two deleted partition. Thus, in each partition merge iteration, we reduce the dimension of our triangle partition matrix by one. We also ensure that we always combine those partitions

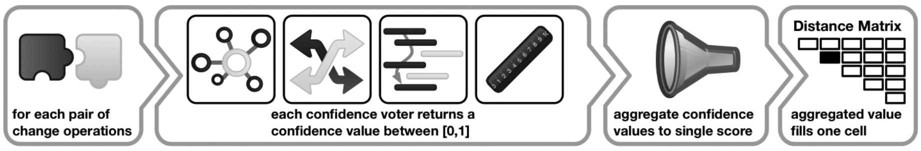

for each pair of change operations
each confidence voter returns a confidence value between [0,1]
aggregate confidence values to single score
Distance Matrix
aggregated value fills one cell

Figure 5.1: The procedure to build the initial triangle matrix used within the modified multilevel graph partitioning algorithm.

that are most likely related to each other. More detailed, the algorithm performs the following steps:

1. Build up a triangle partition matrix $\mathcal{M}$ of dimension $m \times m$ containing one row and one column for each change set partition. We start with the finest granular partitioning of the original change set—one partition for each change operation applied by the change set.

2. For each matrix cell $[P_i, P_j]$ with $i < j \leq m$ of $\mathcal{M}$, we compute a *confidence value* indicating the likelihood that the partitions P_i and P_j are related and should be unified (see Section 5.4 for details on how to compute these confidence values). The confidence value for matrix cell $[P_i, P_j]$ equals the confidence value for the partition pair (P_j, P_i). Figure 5.1 shows this step in detail.

3. After building up the triangle matrix, we determine the pair (P_s, P_t) of partitions with the highest confidence value but for which $s \neq t$. We then delete the two rows and two columns corresponding to the partitions P_s and P_t and add one column and one row and one column for the partition P_{m+1}. The new partition P_{m+1} contains the union of change operations from P_s and P_t. In other words we combine those partitions most likely being related.

4. For the just newly generated partition P_{m+1} we compute confidence values between the new partition and all remaining partitions within $\mathcal{M}$. There are different strategies to compute the confidence values between two partitions P_x and P_y containing multiple change operations. For the presented results, we took the maximum of all confidence values between change operations stemming from different partitions:

$$Conf(P_x, P_y) = Max\{Conf(c_i, c_j) \mid c_i \in P_1 \ \wedge \ c_j \in P_2\}.$$

The intention to use the maximum value here is that two change set partitions can be related even though they have very few properties in common but operate on the same data structure or object instance. In such cases, using the mean value instead would disregard such dependencies.

Without determining a stopping criterion, this algorithm would run until only one partition is left. This would be the original change set itself. Our algorithm can handle two different stopping strategies, both used for different purposes. Giving the algorithm a fixed number of partitions to be produced, it merges partitions until it reaches the desired number of partitions and returns. This strategy might be a good candidate for analyzing change sets of projects that follow a very strict commit message format. In these cases, it might suffice to scan the commit message to extract the expected number of partitions. If the number of partitions to be expected is unknown, the algorithm allows the user to specify a *confidence threshold* that must be exceeded in order to allow to partitions to be merged. If no cell within $\mathcal{M}$ exceeds this threshold, the algorithm terminates.

The untangling algorithm shown so far represents the partitioning framework used to merge change operations into partitions only. This part of the algorithm is general and makes no assumptions about source code, change operations, or any other aspect that estimates the relation between individual change operations. It is important to notice that the generated partitions do not overlap. A change operation can only belong to one partition and there exists no partition belonging to no partition.

In the next sections, we will discuss how to derive the initial confidence values between change operations filling the cells of the initial triangle matrix $\mathcal{M}$ and how to combine multiple dependency measures into a single confidence value.

5.4 Confidence Voters

To combine various dependency and relation aspects between change operations, the untangling framework itself does not decide which change operations are likely to be related but asks a set of so called *confidence voters* (ConfVoters) (see Figure 5.2). Each ConfVoter expects a pair of change operations and returns a *confidence value* between zero and one. A confidence value of one represents a change operation dependency aspect that strongly suggests to put both change operations into the same partition. Conversely, a return value of zero indicates that the change operations are unrelated according to this voter.

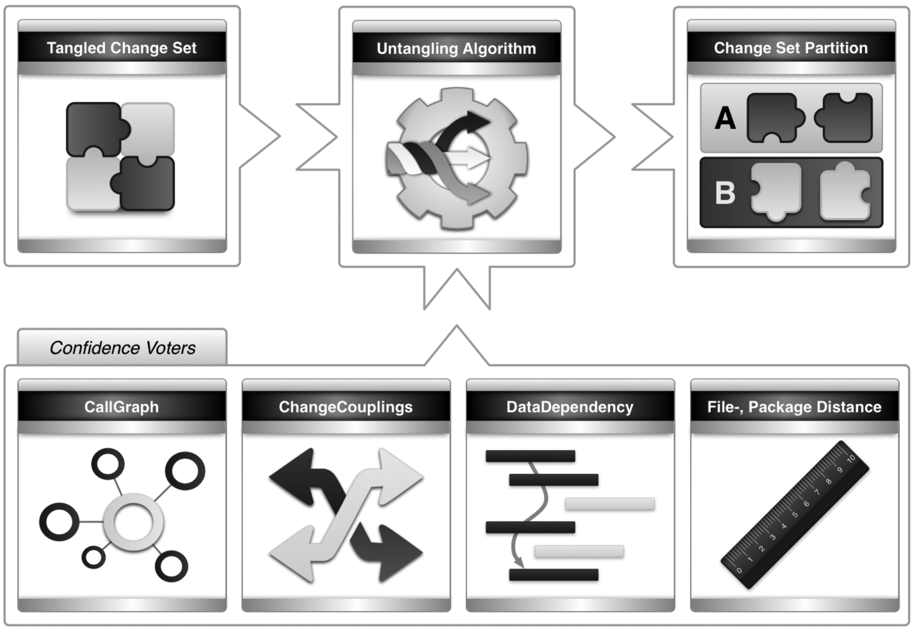


Figure 5.2: The untangling algorithm partitions change sets using multiple, configurable aspect extracted from source code.

ConfVoters can handle multiple relation dependency aspects within the untangling framework. Each ConfVoter represents exactly one dependency aspect. Below we describe the set of ConfVoters used throughout our experiments.

FileDistance In Section 5.3.1 we discussed that change operations are bound to single lines. This ConfVoter returns the number of lines between the two change operation divided by the line length of the source code file both change operations are applied to. If both change operations were applied to different files this ConfVoter will not be considered.

PackageDistance If both change operations were applied to different code files, this ConfVoter will return the number of different package name segments comparing the package names of the changed files. Will not be considered otherwise.

CallGraph Using a static call graph derived after applying the complete change set we identify the modified method definitions and calls and measure the call distance between two call graph nodes. The call graph distance between two method change operations is defined as the sum of all edge weights of the shortest path between both nodes. An edge weight between method m_1 and method m_2 is defined as one divided by the number of method calls between m_1 and m_2.

ChangeCouplings The confidence value returned by this ConfVoter is based on the concept of change couplings as described by Zimmermann et al. [154]. The ConfVoter computes frequently occurring sets of code artifacts that got changed within the same change set. The more frequent two files changed together, the more likely it is that both files are required to be changed together. The confidence value returned by this ConfVoter indicated the probability that the change pattern will occur whenever one of the patterns components change.

DataDependency Returns a value of one if both changes read or write the same variable(s); returns zero otherwise. This relates to any JAVA variable (local, class, or static) and is derived using a static, intra-procedural analysis.

5.5 Evaluation Setup

The current prototype of the untangling framework allows ConfVoters to be registered as plug-ins. This way, we can add or remove project-specific change operation dependency aspects within minutes.

To transform multiple confidence values—one per registered ConfVoter—into a single confidence value required for the initial triangle matrix $\mathcal{M}$ (see Section 5.3.2), we have to aggregate, respecting the fact that different ConfVoters may have different importance (e.g. data dependency might be a stronger indication than change couplings). For this purpose, we train a *linear regression* model to determine a project's specific linear combination of dependency aspects that matter to separate related from unrelated change operations. Once such a model is trained, we can use it to determine a single confidence value (regression) providing all confidence values of all registered ConfVoters. For details on how to train the linear regression model see Section 5.5.3.

To determine the precision of our untangling algorithm, we ran experiments on five open-source Java projects (see Table 5.2). For all projects we analyzed more than 50 months of active development history. Each project counts more than 10 active developers. The number of committed change sets ranges from 1,300 (JAXEN) to 16,000

Table 5.2: Details of projects used during untangling experiments. Comparing to Table 5.1 most change sets could not be classified as atomic or tangled. These change sets are classified as *undecided* and will not be used.

	ARGOUML	**GWT**†	**JAXEN**	**JRUBY**	**XSTREAM**
Lines of code	164,851	266,115	20,997	101,799	22,021
Development history (months)	150	54	114	105	90
# Developers	50	120	20	67	12
# Change sets	16,481	5,326	1,353	11,134	1,756
# Bug fixes	2,945	809	105	2,977	312
# Atomic bug fixes	125 (4.2%)	44 (5.4%)	32 (30.5%)	200 (6.7%)	40 (12.8%)

†GOOGLE WEBTOOL KIT

(ARGOUML), and the number of bug fixing change sets ranges from 105 (JAXEN) to nearly 3,000 (ARGOUML and JRUBY). Choosing projects with different size (# change sets) allow us to check whether the proposed untangling algorithm works on smaller projects as well as on large project histories.

In the next sections, we will discuss how to determine the expected outcome of an untangling procedure in order to measure its precision using a set of ground truths and a method to produce a set of artificially tangled change sets.

5.5.1 Ground Truth

In Section 5.2, we discussed that a significant proportion of change sets must be considered as tangled. For our experimental setup this means that we cannot rely on the existing data to evaluate our untangling algorithm, simply because we cannot determine whether a produced change set partition is correct and if not, how much it differs from an expected result.

We consider change sets that contain only change operations to fulfill exactly one developer maintenance task as atomic and unbiased. To determine a reliable set of atomic and unbiased change sets we used a two-phase manual inspection of issue fixing change sets:

1. We pre-selected change sets that could be linked to exactly one fixed and resolved bug report (similar to Zimmermann et al. [153]).

2. Each change set from Step 1 was manually inspected and classified as atomic or non-atomic. During manual inspection, we first read the commit message. In many cases, the commit message already indicated a tangled change set and therefore the change set was marked non-atomic. If the commit message did not classify the change set as non-atomic, we inspected the actual code changes. Only if we had no doubt that the change set served only one developer maintenance task that was stated within the linked bug report (also no additional refactoring or code cleanup), we classified the change set as atomic. During classification, we tried to be as conservative as possible. If we had any doubt that the change set might not be atomic, we classified it as non-atomic.

The limitation to change sets referencing issue reports was necessary in order to understand the reason of the applied code changes. Without having a document describing the applied changes, it is very hard to judge whether a code change is tangled or not.

The last row of Table 5.2 contains the number of manually classified atomic change sets per project. The number depends on two project specific factors: the more bug fixing change sets contain a reference to the corresponding bug report, the more change sets rank for manual inspection; second, the smaller and cleaner a change set the easier its manual classification is. For the three projects with the lowest number of change sets GOOGLE WEBTOOL KIT, JAXEN, and XSTREAM, we found between 30 and 40 atomic changes each. Considering the total amount of appliedchange sets, this number is small but does not necessarily represent the amount of atomic change sets within the project. Due to the very restrictive selection of change sets for manual classification, we have limited the number of atomic changes to a small selection. For the two larger projects ARGOUML and JRUBY, we found more atomic change sets. The the fraction of atomic change sets (0.8–2.4%) represent the fraction of change sets that could be *classified as atomic* but does not include all atomic change sets. For the vast majority of change sets we could not determine whether these change sets are atomic or tangled. The figures presented in Table 5.1 are the fraction of change sets that could be *classified as tangled*. There exist more tangled change sets but we could not identify them. This implies that the majority of change sets could not be classified by our conservative approach.

5.5.2 Artificial Tangled Change Sets

Evaluating our untangling approach requires a set of tangled change sets for which we already know the correct partitioning. But getting such a set is hard. Tangled change sets get not untangled so far and in many cases a “correct” untangling might not even be possible. Still, we need a set of known and already untangled tangled change sets for evaluation purposes.

There are two main strategies to get such a set of *ground truth*. The first option is to manually untangling natural occurring tangled change sets. But this requires detailed project and source code knowledge and a detailed understanding of the intention behind all change operations applied within a change set. As project outsiders we are not the right persons to perform such a manual untangling and all wrongly partitioned tangled change sets added to the set of ground truth would bias our evaluation set. In this work, we follow the second strategy to generate a set of ground truth: we artificially tangle atomic change sets into *artificially tangled change sets*. For such artificially tangled change sets we know already the perfect partitioning: all change operations applied by the same atomic change set should be put into the same partition. We agree that artificially tangled change sets do not necessarily simulate all types of naturally occurring tangled change sets, but we believe that using artificially tangled change sets for evaluation purposes allows us to demonstrate the general untangling performance of our algorithm and its impact on quality models.

Combining atomic change sets into artificially tangled change sets is straight forward. Nevertheless, we have to be careful which atomic change sets to tangle. Combining them randomly is easy but would not simulate real tangled change sets. In most cases, developers do not combine arbitrary changes, but code changes that are close to each other (e.g. fixing two bugs in the same file or improving a loop while fixing a bug). To simulate such relations to some extend, we combined change sets using the following three tangling strategies:

Change close packages (pack). Using this strategy we combine only change sets that contain at least two change operations touching source files that are not more than two sub-packages apart.

As an example, assume we have a set of three change sets changing three classes $CS_1 = \{com.my.foo.intern.F_1\}$, $CS_2 = \{com.my.foo.extern.F_2\}$, and $CS_3 = \{com.your.foo.intern.F_3\}$. Each class is identified by its fully qualified name. Using this strategy we combine CS_1 with CS_2 since they are only one sub-package apart. But we would not combine CS_1 with CS_3 nor CS_2 with CS_3.[1]

[1]This slightly penalizes the ConfVoter which uses package distances as a heuristic. However, we favored a more realistic distribution of changes over total fairness across all ConfVoters.

Frequently changed before (`coupl`). This strategy computes and uses *change coupling rules* [154] (see also Chapter 6). Two code changes get only tangled if in history at least two code artifacts changed by different change sets showed to be frequently changed together.

For example, let CS_i and CS_j be a pair of atomic change sets and let CS_i be applied before CS_j. CS_i changed file F_s while CS_j changes file F_t. First, we compute all change coupling rules using the approach of Zimmermann et al. [154] and call this set $\mathcal{S}$. The computed change coupling rules indicate how frequently F_s and F_t got changed together in one change set before CS_i got applied. We combine CS_i and CS_j only if $\mathcal{S}$ contains a file coupling rule showing that F_s and F_t had been changed in at least three change sets applied before CS_i. Further we require that in at least 70% of change sets applied before CS_i that changes either F_s or F_t the corresponding other file got changed as well.

Consecutive change sets (`consec`). We combine consecutive change sets applied by the same author. Consecutive change sets are change set that would have ended up in a tangled change set if the developer forgot to commit the previous change set before starting a new developer maintenance task.

Furthermore, we limit all strategies to combine only atomic change sets that lie no more than 14 days apart. The main reason for this limitation was to simulate real world code changes. This limitation was also necessary for technical reasons. Most of our ConfVoters require type resolution using the *partial program analysis* tool [37] that needs to partially compile the source code. Longer time periods between atomic change sets imply higher probability that merging change sets will lead to uncompilable code. Applying our approach to real world tangled change sets, such a situation will never occur.

5.5.3 Training a Confidence Voter Model

In Section 5.4, we mentioned the use of a linear regression model to aggregate all ConfVoter confidence values for a pair of change operations into a single confidence value.

To train the model, we need a set of positive and negative samples. Positive samples are pairs of change operations that belong to the same change set partition. A negative sample consists of change operations belonging to different change set partitions. Positive samples can be directly extracted from our ground truth set containing

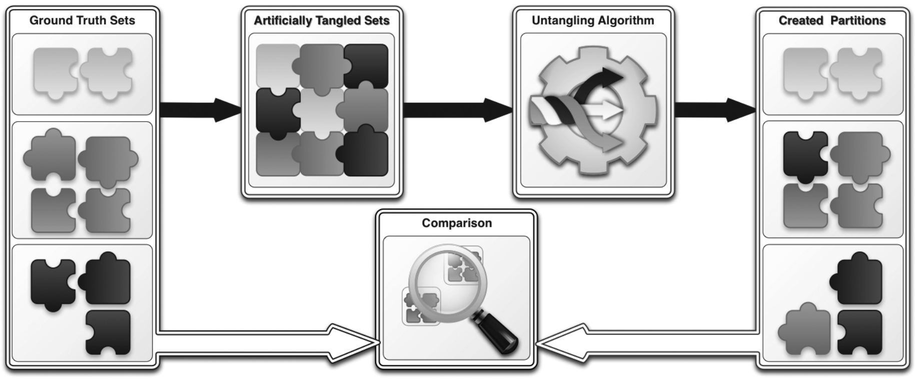


Figure 5.3: Artificially tangled change sets are generated using manually classified atomic change sets to compare created partitions and desired output. Here, two change operations are put into a wrong partition, and hence the precision is $\frac{7}{9} = 77.\overline{7}\%$.

atomic change sets (see Section 5.5.1). Each pair of a change set within the ground truth set is a positive sample. Negative samples can be generated by computing ConfVoter values for pairs of change operations that were applied by different change set from the ground truth set. We use thirty per cent of the corresponding ground truth set to generate positive and negative samples.

Using these positive and negative samples, we will then train our aggregation model using the `Weka` framework [143]. To aggregate confidence values within the untangling algorithm itself, it suffices to let the model predict the result value based on the ConfVoter confidence values computed for each pair of change operations.

5.5.4 Precision

To test our untangling algorithm, we generate all possible artificially tangled change sets as described in Section 5.5.1. Since we know the origin of each change operation, we now can compare the expected partitioning with the partitioning produced by the untangling algorithm (see Figure 5.3). We measure the difference between original and produced partitioning as the number of change operations that were put into a "wrong" partition.

For a set of tangled change sets $\mathcal{B}$, we define *precision* as

$$precision = \frac{\text{\# correctly assigned change operations}}{\text{total \# of change operations} \in \mathcal{B}}$$

As an example for precision, consider Figure 5.3. In the tangled change set, we have 9 change operations overall. Out of these, two are misclassified (the black one in the middle partition, and the gray one in the lower partition); the other seven change operations are assigned to the correct partition. Consequently, the precision is $7/9 = 77.\overline{7}\%$, implying that $2/9 = 22.\overline{2}\%$ of all changes need to be recategorized in order to obtain the ground truth partitioning.

For each set of tangled change sets there exist multiple precision values. The precision depends on which change set partition is compared against which original atomic change set. Precision values reported in this chapter correspond to the partition change set association with the highest sum of *Jaccard indices* [76]. The Jaccard index between two sets expresses the similarity for two sets and is defined as

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

The higher the Jaccard index the higher the similarity of the sets A and B. Thus, by maximizing the sum of Jaccard indices over a set of association permutations relating partitions with atomic change sets we chose the association permutation with the highest similarity of associated pairs. Short, we report the best precision value over all existing association permutations.

5.6 Results and Discussion

The results presented in this section depend on the size of the artificial tangled change sets that is to be untangled—the so called *blob size*. The blob size represents the number of atomic change sets contained within the same artificial tangled change set. We did not generate artificially tangled change sets larger than four, although possible. During our manual change set inspection (see Section 5.2) we rarely found commits that had or exceeded a blob size of four. The distribution of blob sizes is shown in Figure 5.4. 73% of all tangled change sets found during manual inspection had a blob size

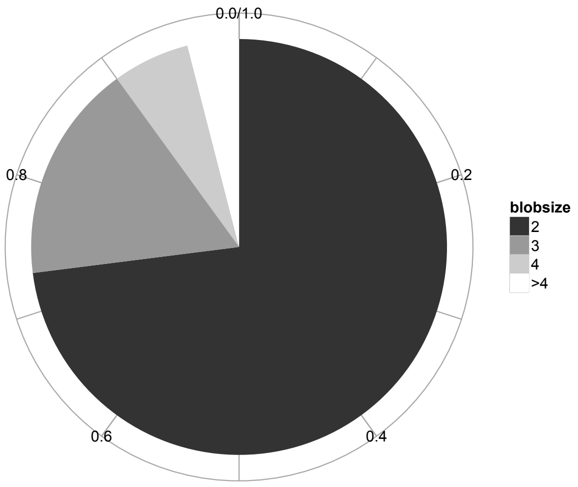


Figure 5.4: The number of tangled change sets discovered during manual inspection having a specific blob size.

of two. Concentrating on tangled change sets with a blob size of four or lower covers 96% of all found tangled change sets. For the presented results, we used a confidence threshold (see Section 5.3.2) of 0.5 to stop further partition aggregation. To determine whether the number of tangled change sets influences the untangling precision (see Section 5.5.4) we report results of different blob sizes separately.

☛ *73% of all discovered tangled change sets had a blob size of two.*

☛ *96% of all discovered tangled change sets had a blob size lower than five.*

5.6.1 Artificially Tangled Change Sets

Table 5.3 contains the number of generated artificially tangled change sets grouped by blob size and combination strategy (see Section 5.5.2). The last three rows of Table 5.3

Table 5.3: Number of generated artificially tangled change sets sorted by blob size and generation strategy. The abbreviation `pack.` stands for the package distance strategy, `coupl.` for change coupling strategy, and `consec.` for the strategy using consecutive change sets.

Blobsize	Strategy	ArgoUML	GWT[†]	JRuby	Xstream
2	`pack.`	40	110	1,430	32
	`coupl.`	0	20	590	0
	`consec.`	180	30	3,364	30
3	`pack.`	13	40	17.3k	133
	`coupl.`	0	0	19.2k	0
	`consec.`	673	70	11.4k	53
4	`pack.`	0	40	1.2M	83
	`coupl.`	0	0	81.9k	0
	`consec.`	743	70	695.3k	25
$\sum$	`pack.`	53	190	1.2M	248
	`coupl.`	0	20	101.1k	0
	`consec.`	1,596	170	710.0k	108

[†]GWT = Google Webtool Kit

contain the sum of artificially tangled change sets generated using different strategies but across different blob sizes. The number of artificially tangled change sets following the change coupling strategy (`coupl.`) is low except for JRuby. The ability to generate artificially tangled change sets from project history depends on the number of atomic change sets, on the number of files touched by these atomic change sets, on the change frequency within the project, and on the number of existing change couplings.

A good example is the Google Webtool Kit project. It is the project with the second lowest number of generated artificially tangled change sets, but also the project with one of the highest fraction of non-atomic bug fixes (see Table 5.1) and the second smallest number of atomic change sets identified (see Table 5.2). The project with the highest number of artificially tangled change sets is JRuby. Accordingly, it is the project with the second lowest number of tangled bug fixes (Table 5.1) and the project with the highest number of atomic change sets (Table 5.2).

Table 5.4: Precision rates of the untangling algorithm sorted by blob size and generation strategy.

Blobsize	Strategy	ArgoUML	GWT[†]	JRuby	XStream	$\overline{x}$
2	pack.	0.79	0.67	0.91	0.81	0.80
	coupl.	—	0.75	0.93	—	0.84
	consec.	0.74	0.70	0.91	0.79	0.79
	$\overline{y}$	0.77	0.71	0.92	0.80	0.80
3	pack.	0.70	0.63	0.69	0.65	0.67
	coupl.	—	—	0.68	—	0.68
	consec.	0.62	0.57	0.70	0.66	0.64
	$\overline{y}$	0.66	0.60	0.69	0.66	0.66
4	pack.	—	0.58	0.62	0.50	0.57
	coupl.	—	—	0.63	—	0.63
	consec.	0.55	0.54	0.64	0.59	0.58
	$\overline{y}$	0.55	0.56	0.63	0.55	0.58

[†]GWT = Google Webtool Kit

5.6.2 Untangling Artificial Tangled Change Sets

The precision of the untangling algorithm is shown in Table 5.4. Precision values are grouped by project, blob size, and tangling strategy. Rows stating $\overline{y}$ as strategy contain the average precision over all strategies for the corresponding blob size. The column $\overline{x}$ shows the average precision across different projects for the corresponding blob generation strategy. The cells $(\overline{x}, \overline{y})$ contain the average precision across all projects and blob generation strategies for the corresponding blob size. Table cells containing no precision values correspond to the combinations of blob sizes and generation strategies for which we were unable to produce any artificially tangled change sets.

The algorithm performs well on all subject projects. Projects with higher number of generated artificially tangled change sets also show higher untangling precision. The more artificially tangled change sets, the higher the number of instances to train our linear regression aggregation model on (see Section 5.5.3). Overall, the precision values across projects show similar ranges and most importantly similar trends in relation to the chosen blob size.

☛ *The untangling algorithm shows comparable results across all subject projects.*

For all projects, the precision is negatively correlated with the used blob size. The more change operations to be included and the more partitions to be generated, the higher the likelihood of misclassifications. Figure 5.4 shows that tangled change sets with a blob size of two are most common (73%). The results in Table 5.4 show that for the most popular cases our untangling algorithm achieves precision values between 0.67 and 0.93—0.80 on average. Increasing the blob size from two to three, precision drops by approximately 14%, across all projects and from 80% to 66% on average. Increasing the blob size further has a negative impact on precision.

☛ *Our approach untangles any two artificially tangled change sets with a precision between 0.67–0.93.*

☛ *The mean precision across all blob sizes is 58% (blob size four) to 80% (blob size two).*

Table 5.4 shows that the precision values for artificially tangled change sets are stable across tangling strategies. For each project and blob size the precision values across different strategies differ at most by 0.09 and on average by 0.04.

☛ *The performance of our untangling algorithm is stable across all tangling strategies.*

In Section 5.1 we discussed the issue of tangled change sets when building defect prediction models. Imagine we want to count the number of applied bug fixes per source file. And lets assume that there exists at least one tangled change set that implements one feature, fixes one bug, and changes two source files. Which file will be assigned a bug fix? Since we do not know which source file had to be changed in order to fix the bug, we can either count one bug for both files or we count one bug for one of the files that we choose randomly. Either strategy may end up with a false positive and thus might introduce noise. Above, we showed that our untangling algorithm is capable of separating change operation applied to fix different code bugs with a reasonable precision. Using our untangling algorithm to increase bug count precision for bug counting model described above, our measured untangling precision does not indicate the fraction of source files that would be assigned a false positive bug fix count.

For this purpose, we measured the *relative file error* during untangling tangled change sets. For each tangled change set untangled we lift the level for computing precision values to the file level measuring the number source code files correctly or

Table 5.5: Relative untangling file error.

Blobsize	Strategy	ArgoUML	GWT†	JRuby	XStream	$\overline{x}$
2	`pack.`	0.00	0.00	0.10	0.30	0.00
	`coupl.`	—	0.25	0.15	—	0.20
	`consec.`	0.26	0.30	0.22	0.32	0.28
	$\overline{y}$	0.13	0.18	0.16	0.31	0.19
3	`pack.`	0.00	0.36	0.32	0.34	0.34
	`coupl.`	—	—	0.42	—	0.42
	`consec.`	0.38	0.40	0.37	0.45	0.40
	$\overline{y}$	0.19	0.38	0.37	0.40	0.29
4	`pack.`	—	0.50	0.47	0.41	0.46
	`coupl.`	—	—	0.48	—	0.48
	`consec.`	0.44	0.40	0.45	0.47	0.44
	$\overline{y}$	0.44	0.45	0.47	0.44	0.45

†GWT = Google Webtool Kit

falsely associated with an untangling result partition. Thus, the relative file error reports the proportion of source files that would be falsely assigned to a developer maintenance task. Table 5.5 shows the results for all subject projects grouped by tangling strategy and blob size. Similar to Table 5.4, the $\overline{y}$ column contains the average error rates over all strategies; $\overline{x}$ rows contain the average error rates across different projects for the corresponding blob generation strategy. The cells $(\overline{x}, \overline{y})$ contain the average error rates across all projects and blob generation strategies for the corresponding blob size.

Overall the average file error rates over all projects and strategies (red colored cells) correspond to the overall precision values in Table 5.4. For a blob size of two, the relative file error rates are below 0.30. Untangling change sets of blob size two reduces the number of source files falsely associated to developer maintenance tasks by at least 70%. Like untangling precision, the relative file error is positively correlated with the blob size. The higher the blob size the lower the untangling precision and the higher the relative file error rates. Although the relative file error rates raise to a value of 0.5 for tangled change sets of size four, we still reduce the number of source files falsely associated to developer maintenance tasks by at least 50%.

☛ *Untangling code changes reduces the number of source files falsely associated to developer maintenance tasks by 55% (blob size four) to 81% (blob size two).*

5.7 Impact on Bug Prediction

To show the impact of untangling tangled change sets we conduct a similar experimental setup as discussed in Section 4.8. We use a simple quality model that identifies the most defect-prone source files by counting the number of distinct bug reports mapped to the corresponding file (see Section 2.3.4).

To compare quality models untangling tangled change sets against classic models, we generate two bug count data sets. For the classic reference models, we associate all referenced bug reports to all source files changed by a change set, disregarding whether we marked it tangled or not. For the untangled bug count set, we used our untangling algorithm to untangle manual classified tangled change sets. If the tangled change set reference bug reports only, we assigned one bug report to each partition—since we only count the number of bug, it is not important which report gets assigned to which partition.

For change sets referencing not only bug reports we used an automatic change purpose classification model based on the findings of Herzig et al. [67] (see also Chapter 6) indicating that bug fixing change sets apply less change operations when compared to feature implementing change sets. Thus, we classify those partitions applying the fewest change operations as bug fixes. Only those files that were changed in the bug fixing partitions were assigned with one of the bug reports. Both bug counting sets get sorted in descending order using the distinct number bug reports associated with the file (see Figure 5.5).

As in Section 4.8 the most defect-prone file is the top element in each bug count set. Comparing the top X% of both file sets (containing the same elements but in potentially different order) allows us to reason about the impact of tangled change sets on models using bug counts to identify the most defect-prone entities. Since both cutoffs are equally large (the number of source files does not change, only their ranks), we can define the *cutoff_difference* as:

$$\frac{\text{size of cutoff} - \text{size of intersection}}{\text{size of cutoff}}.$$

The result is a number between zero and one where zero indicates that both cutoffs are identical and where a value of one implies two cutoffs with an empty intersection. A low cutoff_difference is desirable.

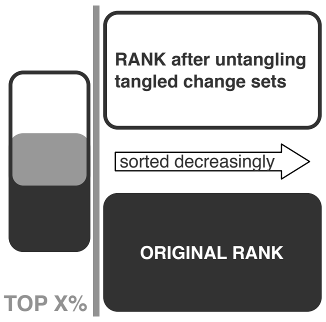


Figure 5.5: The cutoff_difference for the top x% illustrating the impact of tangled change sets on quality data models.

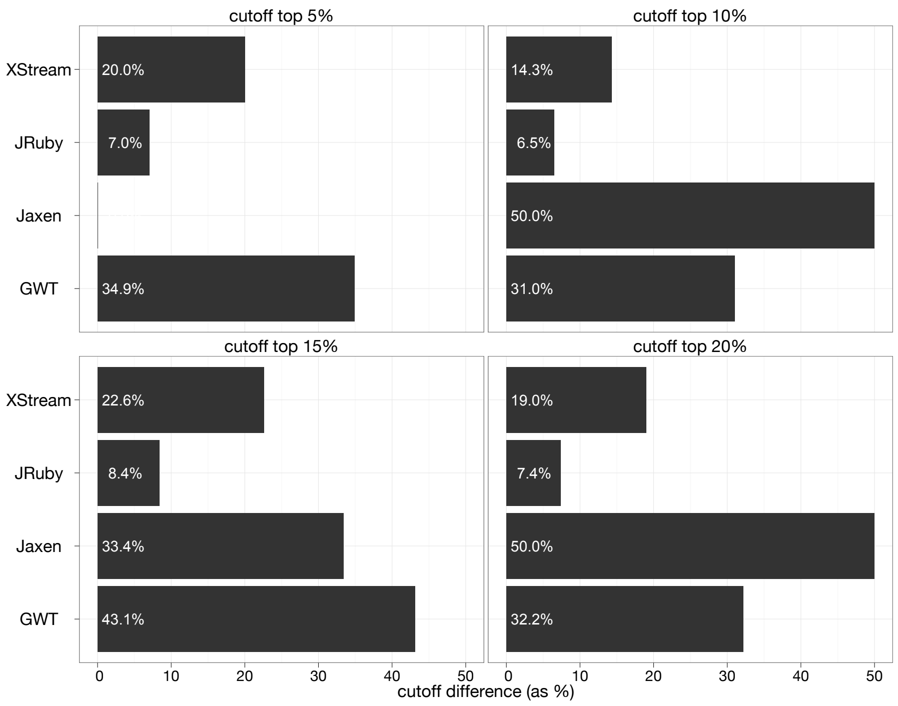


Figure 5.6: The cutoff_differences caused by tangled change sets.

Remember that we untangled only those change sets that we manually classified as tangled change sets (see Table 5.1). The fraction of tangled change sets lies between 6% and 15%, only. Figure 5.6 shows that untangling these few tangled change sets already has a significant impact on the set of source files marked as most defect prone. Figure 5.6 shows the cutoff_differences for the top 5%, 10%, 15%, and 20% of files with the highest distinct number of associated bug reports. The impact of untangling lies between 0% and 50%. It is not surprising that cutoff_difference and fraction of tangled change sets is correlated. JRuby had the lowest fraction of blobs and shows the smallest cutoff_differences. Jaxen had the highest tangled change set fraction and shows the highest cutoff_differences. We can summarize that untangling tangled change sets impacts bug counting models and thus are very likely to impact more complex quality models or even bug prediction models trained on these data sets, too.

☛ *Tangled change sets severely impact bug counting models. On our open-source projects, untangling manual classified tangled change sets showed that between 6% and 50% of the most defect prone files do not belong in this category because they were falsely associated with bug reports.*

We further observed that in total between 10% and 38% of all source files we assigned different bug counts when untangling tangled change sets. Between 1.5% and 7.1% of the files originally associated with bug report had no bug count after untangling tangled change sets.

5.8 Threats to Validity

Like any other empirical study of this kind, the approach presented in this study has threats to its validity. We identified four noteworthy threats.

The change set classification process used in Section 5.5.1 involved manual code change inspection. We tried to be as conservative as possible when classifying atomic change sets, but the classification process was conducted by software engineers not familiar with the internal details of the individual projects. Thus, it is not unlikely that the manual selection process or the pre-filtering process misclassified change sets. This could impact the number and the quality of generated artificially tangled change sets and thus could impact the untangling results, in general.

A second threat is given by the selected software projects used throughout our experimental setup. We cannot claim that the selected JAVA projects are representative in any way. We tried to include projects of different size and domains. With GOOGLE WEBTOOL KIT, we also considered a project that is developed by an industrial company. Nevertheless, we have to be aware that untangling results for other projects may differ. The very same holds for the selection of ConfVoters. Choosing a different set of ConfVoters will impact untangling results.

The process of constructing artificially tangled change sets may not be simulating real life tangled change sets caused by developers combining multiple developer maintenance task into single change sets. Thus, results of untangling real developer change sets may differ.

The untangling results presented in this paper are based on artificially tangled change sets derived using the ground truth set which contains issue fixing change sets, only. Thus, it might be that the ground truth set is not representative for all types of code changes.

The aggregation process to transform multiple confidence values into a one includes a machine learner training phase. The data sets used to train these aggregation models are produced by random splits (see Section 5.5.3). Using different random splits may impact the aggregation results significantly and thus may impact the overall untangling results.

We use bug counting models to measure the impact of untangling on defect prediction and other code quality related models. Although most defect prediction models use these simple bug count models as underlying classification or regression basis, it might be that the impact of tangled change sets on real prediction models differs from the presented results.

As mentioned briefly, we use the partial program analysis tool [37] by Dagnais and Hendren within our untangling algorithm. Thus, the validity of our results depends on the validity of the used approach.

5.9 Summary

This work proposes an untangling algorithm that helps to reduce the amount of bias within data mining sets, caused by version control system commits. Combining code changes serving multiple developer maintenance tasks into one change set produces tangled change sets. Our results support the findings of Kawrykow [83] and show that the fraction of tangled changes may be substantial, causing a serious threat to empirical findings based on version archives.

For the five open-source projects used within our experiments, our untangling algorithm showed an average precision between 58% and 80%, depending on the number of developer maintenance tasks combined in a single change set. Basing empirical findings on untangled changes will make them more precise, and less threatened by bias and noise. Independent results reported by Kawrykow [83] show similar precision rates.

We are committed to make the entire untangling framework and all training examples publicly available. We expect to release this package early 2013 at the project web site:

`http://www.softevo.org/untangling`

Chapter 6

Classifying Code Changes and Predicting Defects Using Change Genealogies

Parts of the contents of this chapter have been published in the technical report Herzig et al. [67] (currently under submission).

6.1 Introduction

Identifying bug fixes and using them to estimate or even predict software quality is a frequent task when mining version archives. The number of applied bug fixes serves as a code quality metric identifying defect-prone and non-defect-prone code artifacts. But when is a change set considered a bug fix and which metrics should be used to build high quality defect prediction models? Most commonly, bug fixes are identified by analyzing commit messages—short, mostly unstructured pieces of plain text. Commit messages containing keywords such as “fix” or “issue” followed by a bug report identifier, are considered to fix the corresponding bug report (see Section 2.3.4). Similar, most defect prediction models use metrics describing the structure, complexity, churn, or dependencies of source code artifacts.

But commit messages and code metrics describe the state of software artifacts and code changes at a particular point in time, disregarding their genealogies that describe how the current state of the source code came to be. There are approaches measuring historic properties of code artifacts [58, 65, 89, 102, 110] and using code dependency graphs [151, 22] but non of these approaches track the structural dependency paths of code changes to measure the centrality and impact of change sets. But change sets are those development events that make the source code look as it does.

In this chapter, we make use of change genealogies to define a set of *change genealogy metrics* (referred to as *CGMs* for sake of brevity) describing the structural dependencies of change sets. We further investigate whether *CGMs* can be used to identify bug fixing change sets (without using commit messages and bug databases) and whether *CGMs* are expressive enough to build effective defect prediction models classifying source files to be defect-prone or not.

Regarding the identification of change sets dedicated to fix bugs, our assumption is that change sets applying bug fixes show significant dependency differences when compared to change sets applying new feature implementations. We suspect that implementing and adding a new feature implies adding new method definitions that impact a large set of later applied code changes, which add code fragments adding method calls to these newly defined methods. In contrast, we suspect bug fixes to be relatively small (as shown by Mockus and Votta [101]) rarely defining new methods but modifying existing features and thus having a small impact on later applied code changes. The impact of bug fixes is to modify the runtime behavior of the software system rather than causing future change sets to use different functionality.

We call change sets having dependencies to a large number of earlier or later applied change sets *central* and suspect such central change sets to be crucial to the software development process. Consequently, we suspect code artifacts that got many crucial and central code changes applied to be more defect prone than others. More specifically, we seek to answer the following research questions in our study:

RQ6.1 How do bug fix classifiers based on change genealogy metrics compare to classification models based on code complexity churn metrics (Section 6.4)?

RQ6.2 How do defect prediction models compare with defect prediction models based on code complexity or code dependency network metrics (Section 6.5)?

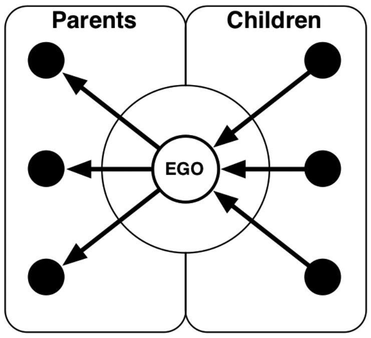
We tested the classification and prediction abilities of our approaches on four open source projects. The results show that *CGMs* can be used to separate bug fixing from feature implementing change sets with an average precision of 72% and an average recall of 89%. Our results also show that defect prediction models based on *CGMs* can predict defect-prone source files with precision and recall values of up to 80%. On average the precision for change genealogy models lies at 69% and the average recall at 81%. Compared to prediction models based on code dependency network metrics, change genealogy based prediction models achieve better precision and comparable recall values.

6.2 Change Genealogy Metrics

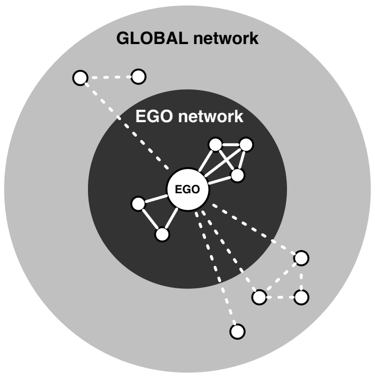
In Chapter 3 we discussed the concept of change genealogies and how to construct them in detail. Summarizing, change genealogies model dependencies (edges) between individual change sets (vertices). Similar to code dependency metrics [151, 22] we can use change genealogies to define and compute change genealogy metrics (*CGMs*) describing the dependency structures between code changes instead of code artifacts. Each change set applied to the software system is represented by a change genealogy vertex. Computing network metrics for each change genealogy vertex means to compute change set dependency metrics. Later, we will use this set of *CGMs* to classify change sets as bug fixing or feature implementing using a machine learner and to predict defect-prone source code artifacts.

In accordance to social network metrics, we divide the neighborhood of a change genealogy vertex called EGO into two main categories: the *ego network* and the *global network* (see Figure 6.1(b)). The ego network contains all direct children and direct parents of the EGO vertex. Further, we distinct between metrics based on structural holes [28], centrality measures, temporal change genealogy metrics, and change set related metrics. We also distinguish between metrics related to change genealogy *parents* and change genealogy children (see Figure 6.1(a)) of the EGO vertex. Parents of EGO are all those vertices that can be reached by following outgoing dependency edges. Code changes represented by the EGO vertex could have not been applied without applying the parent code changes before. Children of a vertex are those vertices that can be reached by following incoming edges. Code changes applied by children depend on code changes applied by EGO.

To capture as many of such dependency differences as possible, we implemented various genealogy dependency metrics of different categories.



(a) Parents and children of a change genealogy vertex.



(b) EGO and global network neighborhoods in change genealogies.

Figure 6.1: Parents, children, global and EGO networks in change genealogies.

6.2.1 EGO Network Metrics

Ego network metrics measure dependencies between change genealogy vertices and their direct neighbors (see Figure 6.1). For every vertex we consider direct children or direct parents, only. Thus, this set of metrics measures the *immediate impact* of change sets on other change sets. Table 6.1 describes the implemented change genealogy ego network metrics.

The metrics *NumDepAuthors* and *NumParentAuthors* refer to the authorship of change set. Bug fixes might depend mainly on change sets that have the same author. The last six metrics in Table 6.1 express temporal dependencies between change sets based on their commit timestamp.

6.2.2 GLOBAL Network Metrics

Global network metrics describe a wider neighborhood (see Figure 6.1). Most global network metrics described in Table 6.2 can be computed for the global universe of vertices and dependencies. For practical reasons, we limited the metric traversal depth to a maximal depth of five.

Table 6.1: Ego network metrics capturing direct neighbor dependencies.

Metric name	**Description**
NumParents	The distinct number of vertices being source of an incoming edge.
NumDefParents	The distinct number of vertices representing a method definition operation and being source of an incoming edge.
NumCallParents	The distinct number of vertices representing a method call operation and being source of an incoming edge.
NumDependants	The distinct number of vertices being target of an outgoing edge.
NumDefDependants	The distinct number of vertices representing a method definition operation and being target of an outgoing edge.
NumCallDependants	The distinct number of vertices representing a method call operation and being target of an outgoing edge.
AvgInDegree	The average number of incoming edges.
AvgOutDegree	The average number of outgoing edges.
NumDepAuthors	The distinct number of authors responsible for the direct children.
NumParentAuthors	The distinct number of authors that implemented the direct ascendants of this vertex.
AvgResponseTime	The average number of days between a vertex and all its children.
MaxResponseTime	The number of days between a vertex and the latest applied child.
MinResponseTime	The number of days between a vertex and the earliest applied child.
AvgParentAge	The average number of days between a vertex and all its parents.
MaxParentAge	The number of days between a vertex and the earliest applied parent.
MinParentAge	The number of days between a vertex and the latest applied parent.

Table 6.2: Global network metrics.

Metric name	**Description**
NumParents[†]	The distinct number of vertices being part of on incoming path.
NumDefParents[†]	Like *NumParents* but limited to vertices that change method definitions.
NumCallParents[†]	Like *NumParents* but limited to vertices that change method calls.
NumDependants[†]	The distinct number of vertices being part of on outgoing path.
NumDefDependants[†]	Like *NumDependants* but limited to vertices that change method definitions.
NumCallDependants[†]	Like *NumDependants* but limited to vertices that change method calls.
NumSiblingChildren	The number of children sharing at least one parent with this vertex.
AvgSiblingChildren	The average number of parents this vertex and its children have in common.
NumInbreedParents	The number of grandparents also being parents.
NumInbreedChildren	The number of grandchildren also being children.
AvgInbreedParents	The average number of grandparents also being parent.
AvgInbreedChildren	The average number of grandchildren also being children.

[†] maximal network *traversal depth* is set to 5.

Metrics counting the number of global descendants or ascendants express the indirect impact of change sets on other change sets and how long this impact propagates though history. The set of inbreed metrics express dependencies between a change set and its children in terms of common ascendants or descendants. Code changes that depend on nearly the same earlier change sets as their children might indicate reverted or incomplete changes.

6.2.3 Structural Holes

The concept of structural holes was introduces by Burt [28] and measures the influence of actors in balanced social networks. In networks where each actor is connected to all other actors is well balanced. As soon as dependencies between individual actors are missing ("structural holes") some actors are in advanced positions (see Figure 6.2).

The *effective size* of a network is the number of change sets that are connected to a vertex minus the average number of ties between these connected vertices. The *efficiency* of a change set is its *effective size* normed by the number of vertices contained in the ego network. The higher the metric values for these metrics the closer the connection of a change set to its ego network. Table 6.3 lists the complete list of used structural hole metrics.

6.2.4 Change Metrics

The last set of metrics shown in Table 6.4 measures the amount of code churn applied by the corresponding change sets and its neighbors in the ego network. In our case, we counted the different number of added and deleted method definitions and method calls. The intuition behind these churn metrics is that bug-fixes should in general be considerable smaller than other developer tasks such as feature implementations or code cleanups [101].

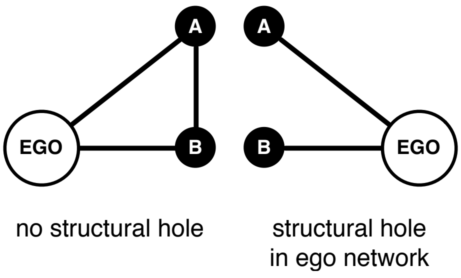


Figure 6.2: The concept of structural holes in ego networks.

Table 6.3: Structural holes metrics similar as defined by Burt [28].

Metric name	Description
EffSize	The number of vertices connected to this vertex minus the average number of ties between these connected vertices.
InEffSize	The number of vertices connected by incoming edges to this vertex minus the average number of ties between these connected vertices.
OutEffSize	The number of vertices connected by outgoing edges to this vertex minus the average number of ties between these connected vertices.
Efficiency	norms EffSize by the number of vertices of the ego-network.
InEfficiency	norms InEffSize by the number of vertices of the ego-network.
OutEfficiency	norms OutEffSize by the number of vertices of the ego-network.

Table 6.4: Change size metrics describe a vertex using the number of change operations applied by the corresponding code change.

Metric name	**Description**
ChangeSize	The number of change operations corresponding to the vertex.
NumAddOps	The number of adding change operations corresponding to the vertex.
NumDelOps	The number of deleting change operations corresponding to the vertex.
NumAddMethDefOps	The number of change operations adding method definitions corresponding to the vertex.
NumDelMethDefOps	The number of change operations deleting method definitions corresponding to the vertex.
NumAddCallOps	The number of change operations adding method calls corresponding to the vertex.
NumDelCallOps	The number of change operations deleting method calls corresponding to the vertex.
AvgDepChangeSize	The mean number of change operations applied by direct children.
MaxDepChangeSize	The maximal number of change operations applied by one of the direct children.
SumDepChangeSize	The total number of change operations applied by direct children.
AvgParentChangeSize	The mean number of change operations applied by direct parents.
MaxParentChangeSize	The maximal number of change operations applied by one of the direct parents.
SumParentChangeSize	The total number of change operations applied by direct parents.

Table 6.5: Projects used for experiments.

	HTTPCLIENT	JACKRABBIT[†]	LUCENE	RHINO
History length	6.5 years	8 years	2 years	13 years
Lines of Code	57,143	65,764	362,128	56,084
# Source files	570	687	2,542	217
# Code changes	1,622	7,465	5,771	2,883
# Mapped BUG reports	92	756	255	194
# Mapped RFE reports	63	305	203	38
		Change genealogy details		
# vertices	973	4,694	2,794	2,261
# edges	2,461	15,796	8,588	9,002

[†] considered only sub-project JACKRABBIT `content repository` (*JCR*).

6.3 Data Collection

The goals of our approach are (a) to classify bug fixing change sets independent from commit messages and bug databases and (b) to predict defect prone source files, both using *CGMs*. To reason about the precision of our classification and prediction models, we compare our derived models to state-of-the-art benchmark models.

Mockus and Votta [101] (referred to as M&V for sake of brevity) used code churn metrics to identify bug fixing change sets. Following their approach, we compute complexity churn metrics to train change purpose classification benchmark models. Section 6.3.2 contains details on the used complexity metrics. Section 6.3.3 describes the used code complexity churn metrics and how to compute them.

We compare change genealogy defect prediction models against two benchmark models: models based on code complexity (see Section 2.3.6) and models based on code dependency metrics as proposed by Zimmermann and Nagappan [151] (referred to as Z&N). Section 6.3.2 and Section 6.3.4 contain details on the used complexity and code dependency network metrics.

We evaluate our classification and prediction models on four open-source projects: HTTPCLIENT, LUCENE, RHINO, and JACKRABBIT. The projects differ in size from small (HTTPCLIENT) to large (LUCENE) allowing us to investigate whether the classification and prediction models are sensitive to project size. All projects are known in the research

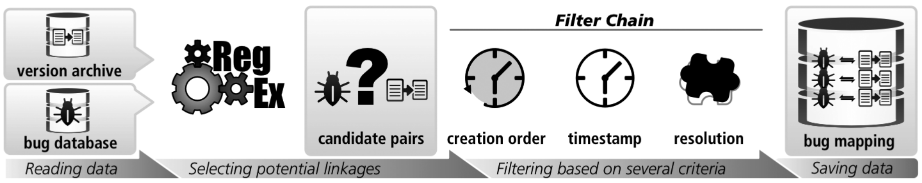


Figure 6.3: Process of linking change sets to bug reports.

community and follow the strict and industry-like development processes of APACHE and MOZILLA. A brief summary of the projects and their genealogy graphs is presented in Table 6.5. Change genealogies contain approximately as many vertices as applied change sets. The difference in the number of vertices and the number of change sets is caused by change sets that do not add or delete any method definition or call (e.g. documentation modifications).

6.3.1 Bugs

For both approaches, change classification and defect prediction, we need to identified bug fixing change sets and the corresponding bug ID. Using this mapping we can associate bug fixing change sets with modified source files and count the distinct number of fixed bug reports per source file.

To associate change sets with bug reports, we followed the approach by Zimmermann et al. [153] (see also Figure 6.3):

1. Bug reports and change sets are read from the corresponding bug tracking system and version archive.
2. In a preprocessing step we select potential candidates using regular expressions such as `[bug|issue|fixed]:?\s*#?\s?(\d+)` to search for potential bug report references in commit messages.
3. The pairs received from step 2) then pass a set of filters checking
   (a) that the bug report is marked as resolved.
   (b) that the change set was applied after the bug report was opened.
   (c) that the bug report was marked as resolved not later than two weeks after the change set was applied.

To determine a set of ground truth identifying the real purpose of change sets we use the manual classified bug report data set described in Chapter 4. This set contains a manual classified issue report type for each individual filed issue report. Instead of using the original issue report type to identify bug reports, we used the manual classified issue report type.

6.3.2 Complexity Metrics (*CMs*)

We computed complexity metrics (*CMs*) for all source files of each projects trunk version using a commercial tool called `JHawk` [2]. `JHawk` computes classical object-oriented code complexity metrics for Java projects. Using `JHawk` we computed the code complexity metrics listed in Table 6.6.

6.3.3 Complexity Churn Metrics (*C$_\Delta$Ms*)

For each change set we compute complexity churn metrics derived as the difference in code complexity before and after the change set was applied. Thus, we compute the set of complexity metrics described in Section 6.3.2 for each historic revision of the software project. For each metric $\mathcal{M} \in CMs$ we sum up the metric values over all source files at revision CS_{i-1} and subtract the same sum of metric values collected at revision CS_i. Doing this for every code complexity metrics, yields a set of code complexity metric values reflecting the amount of code complexity added or deleted by change set CS_i—we call this set *code complexity churn metrics* (*C$_\Delta$Ms*).

6.3.4 Network Metrics (*NMs*)

Code dependency network metrics as proposed by Z&N express the information flow between code entities modeled by code dependency graphs. The set of network metrics used in this work slightly differs from the original metric set used by Z&N. We computed the used network metrics using the *R statistical software* [120] and the *igraph* [36] package. Using *igraph* we could not re-implement two of the 25 original network metrics: *ReachEfficiency* and *Eigenvector*.

Table 6.6: Set of code complexity metrics used.

Identifier	**Description**
NOM	Total number of methods per source file.
LCOM	Lack of cohesion of methods in source file.
AVCC	Cyclomatic complexity after McCabe [96].
NOS	Number of statements in source file.
INST$^{\Sigma}$	Number of class instance variables.
PACK	Number of imported packages.
RCS$^{\oslash}$	Total response for class (# methods + # distinct method calls).
CBO$^{\oslash}$	Couplings between objects [33].
CCML	Number of comment lines.
MOD$^{\oslash}$	Number of modifiers for class declaration.
INTR$^{\Sigma}$	Number of implemented interfaces.
MPC$^{\oslash}$	Represents coupling between classes induced by message passing.
NSUB$^{\Sigma}$	Number of sub classes.
EXT$^{\Sigma}$	Number of external methods called.
FOUT$^{\Sigma}$	Also called *fan out* or *effect coupling*. The number of other classes referenced by a class.
F-IN$^{\Sigma}$	Also called *fan in* or *afferent coupling*. The number of other classes referencing a class.
DIT$^{\wedge}$	The maximum length of a path from a class to a root class in the inheritance structure.
HIER$^{\Sigma}$	Number of class hierarchy methods called.
LMC$^{\Sigma}$	Number of local methods called.

Σ aggregated using the sum of all metric values of lower order granularity.

$\oslash$ aggregated using the mean value.

$\wedge$ aggregated using the max value.

While we simply excluded *ReachEfficiency* from our network metric set, we substituted the *Eigenvector* by *alpha.centrality*—a metric that can be "considered as a generalization of eigenvector centrality to directed graphs" [25]. Table 6.7 lists all code dependency network metrics used in this work. Metrics carry the same metric name than the corresponding metric described by Z&N. For the sake of brevity, we refer the set of network metrics shown in Table 6.7 as *NMs*.

6.3.5 Genealogy Metrics (*CGMs*)

We discussed the set of genealogy metrics in Section 6.2. To compute these metrics, we constructed change genealogy graphs modeling dependencies between change sets over the entire project history.

6.4 Classifying Code Changes (RQ6.1)

In this first series of experiments we seek an answer to *RQ6.1*: Can we use change genealogy metrics to identify bug fixing change sets and how do such code change classification models compare to classification models based on *$C_\Delta Ms$*?

For each subject project, we build two sets of change set classification models and compare both sets of classification models against each other. For each classification model to be built, we need a data collection containing explanatory variables (metric values per change set) and the dependent variable classifying the corresponding change sets as bug fixing or as feature adding (see Figure 6.4). The columns containing *CGMs* are used to train the change genealogy classification model, the code complexity churn columns are used to train the benchmark model.

6.4.1 Experimental Setups

We use two different experimental setups to evaluate the change set classification power of *CGMs*. We first conduct a stratified repeated holdout setup to measure the classification quality within each individual subject project. We then conduct a cross-project classification setup to see whether *CGMs* can be used to train more universal change set classification models sharing a common set of the most influential metrics.

Table 6.7: List of code dependency network metrics.

Metric name	**Description**
Ego-network metrics (computed each for incoming, outgoing, and undirected dependencies; descriptions adapted from Z&N):	
Size	# nodes connected to the ego network
Ties	# directed ties corresponds to the number of edges
Pairs	# ordered pairs is the maximal number of directed ties
Density	% of possible ties that are actually present
WeakComp	# weak components in neighborhood
nWeakComp	# weak components normalized by size
TwoStepReach	% nodes that are two steps away
Brokerage	# pairs not directly connected. The higher this number, the more paths go through ego
nBrokerage	Brokerage normalized by the number of pairs
EgoBetween	% shortest paths between neighbors through ego
nEgoBetween	EgoBetween normalized by the size of the ego network
Structural metrics (descriptions adapted from Z&N):	
EffSize	# entities that are connected to an entity minus the average number of ties between these entities
Efficiency	Normalizes the effective size of a network to the total size of the network
Constraint	Measures how strongly an entity is constrained by its neighbors
Hierarchy	Measures how the constraint measure is distributed across neighbors. When most of the constraint comes from a single neighbor, the value for hierarchy is higher
Centrality metrics (computed each for incoming, outgoing, and undirected dependencies; descriptions adapted from Z&N):	
Degree	# dependencies for an entity
nDegree	# dependencies normalized by number of entities
Closeness	Total length of the shortest paths from an entity (or to an entity) to all other entities
Reachability	# entities that can be reached from a entity (or which can reach an entity)
alpha.centrality†	Generalization of eigenvector centrality [25]
Information	Harmonic mean of the length of paths ending in entity
Betweenness	Measure for a entity in how many shortest paths between other entities it occurs
nBetweenness	Betweenness normalized by the number of entities

† Metrics not used by Z&N.

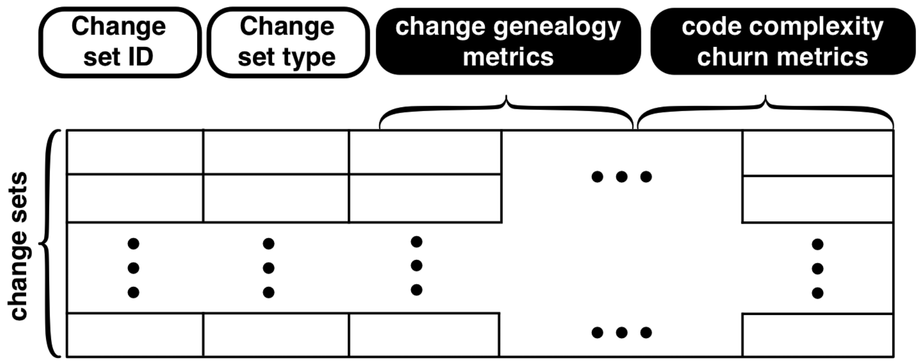


Figure 6.4: Data collection used for change set classification purposes.

We conducted our experiments using the *R statistical software* [120] and more precisely Max Kuhn's R package *caret* [90]. This package provides helpful wrapper functions to several machine learning algorithms available in other packages. Table 6.8 lists the prediction models we used for classification. Each model can be optimized using several different parameters. This is handled by the *caret* package when a *tuneLength* value is specified. We set this number to five.

As evaluation measures, we report *precision*, *recall*, and *F-measure*. Each of these measures is a value between zero and one. A precision of one indicated that the classification model did not produce any false positives; that is classified non bug fixes as bug fixes. A recall of one would imply that the classification model did not produce any false negatives—classified a bug fix not as such. The F-measure represents the harmonic mean of precision and recall.

Stratified Repeated Holdout Setup

To train and test our classification models on each subject project, we split our original data set as shown in Figure 6.4 into training and testing subsets using stratified sampling——the ratio of bug fixing change sets in the original data set is preserved in both training and testing data sets. This makes training and testing sets more representative by reducing sampling errors.

Next, we split the training and testing sets into subsets. Each subset contains the change set type columns, change set type, and identifier columns. One subset contains

Table 6.8: List of models used for classification experiments.

Model[*]	**Description**
k-nearest neighbor (*knn*)	*This model finds k training instances closest in Euclidean distance to the given test instance and predicts the class that is the majority amongst these training instances.*
Logistic regression (*multinom*)	*This is a generalized linear model using a logic function and hence suited for binomial regression, i.e. where the outcome class is dichotomous.*
Recursive partitioning (*rpart*)	*A variant of decision trees, this model can be represented as a binomial tree and popularly used for classification tasks.*
Support vector machines (*svmRadial*)	*This model classifies data by determining a separator that distinguishes the data with the largest margin. We used the radial kernel for our experiments.*
Tree Bagging (*treebag*)	*Another variant of decision trees, this model uses bootstrapping to stabilize the decision trees.*
Random forest (*randomForest*)	*An ensemble of decision tree classifiers. Random forests grow multiple decision trees each "voting" for the class on an instance to be classified.*

[*] For a fuller understanding of these models, we advise the reader to refer to specialized machine learning texts such as by Wittig and Frank [143].

only *CGMs* and one subset contains only $C_\Delta Ms$. Splitting metric sets after creating testing and training sets, we create pairs of classification models using the same training and testing split but using different metrics data as feature vectors.

We repeatedly sample the original data sets 100 times in order to generate 100 independent training and testing sets. Each split is used to built one change genealogy and one code complexity churn model. In total, we test 200 independent prediction models for each project.

Cross-Project Setup

The cross project classification setup entails using data sets from one project to classify change sets of another project. The rationale behind evaluating this setup is to verify whether change set classification models are transferable from one project to another. If the results are promising, it will suggest that projects with little or no data from the past can leverage data from other projects for classification purposes.

Additionally, we performed a stratified repeated holdout experiment using the metric sets over all four subject projects. Achieving good classification results with such a setup suggest universal change set properties that can be used for change set classification purposes.

6.4.2 Classification Quality

Stratified Repeated Holdout Setup

The results of the stratified repeated holdout setup are shown in Figure 6.5. Panels on the x-axis represent the subject projects. Each classification model ran on 100 stratified random samples on the two metric sets: change genealogy and complexity difference metrics.

The black line in the middle of each boxplot indicates the median value of the distribution. The red colored horizontal lines do not have any statistical meaning—they have been added to ease visual comparison. Additionally, we performed a non-parametric statistical test (Kruskal-Wallis) to statistically compare the results from the use of two pairs of metrics sets: change genealogy metrics vs. code complexity metric differences.

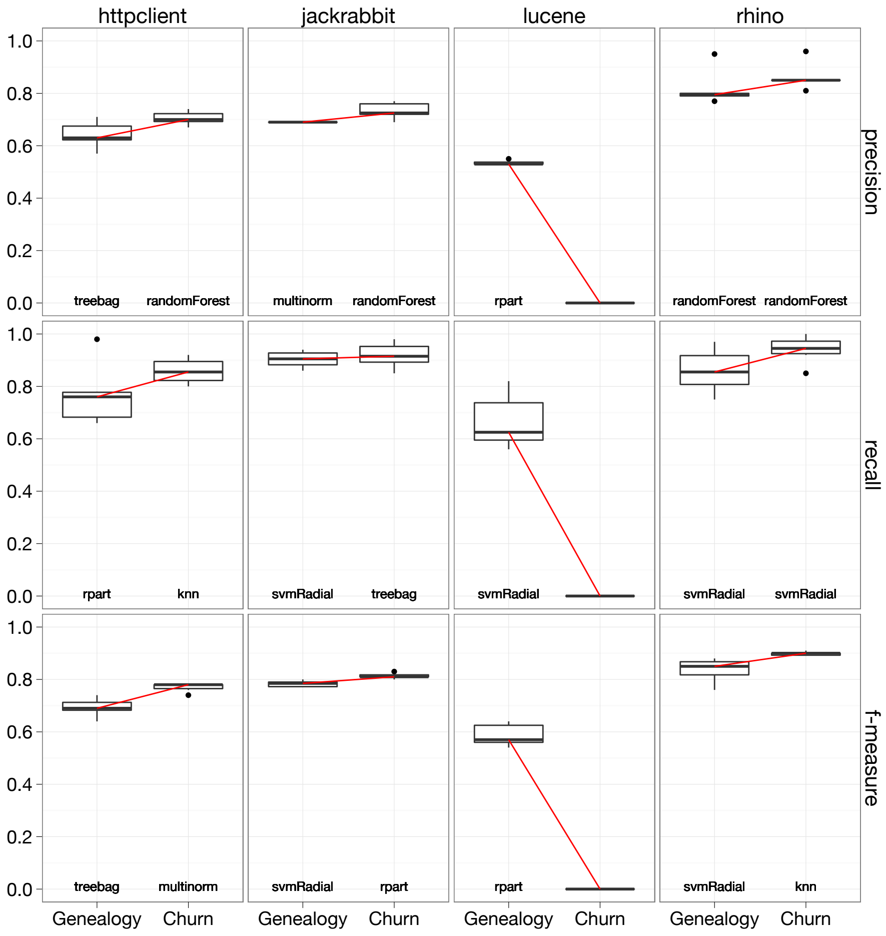


Figure 6.5: Results from the repeated holdout experiment to separate bug fixing from feature implementing code changes.

The results shown in Figure 6.5 show that the classification performances of both metric sets are close to each other, except for LUCENE. In all three cases $C_{\Delta}Ms$ show stronger classification results than *CGMs*. In fact, the statistical tests showed that the difference in classification performance is statistically significant ($p < 0.05$) except for the recall values for JACKRABBIT. In summary, $C_{\Delta}Ms$ outperform *CGMs* on three out of four projects while for LUCENE $C_{\Delta}Ms$ could not be used to train a functional bug fix classification model. Nearly all LUCENE change sets modified code complexity only marginally. Thus, $C_{\Delta}Ms$ showed too little variance to allow classification model training. Project size seems to have no impact on classification accuracy.

Over all projects, classifiers based on *CGMs* showed a median precision of 0.69 and a median recall of 0.81. Models based on complexity churn showed a median precision of 0.72 and a median recall of 0.89. Figure 6.5 also shows that different machine learning models give best results for different projects, metric sets, and evaluation measures.

- ☛ *In 3 of 4 cases complexity metrics show statistically significant better performance measures when compared to change genealogy metrics.*
- ☛ *Classification models based on change genealogy metrics show a median precision of 0.69 and a median recall of 0.81.*
- ☛ *Over all projects classification models based on complexity metrics perform slightly better than models based on change genealogy metrics.*

Cross-Project Setup

The results from the cross-project experimental setup are shown in Figure 6.6. The panels across the x-axis indicate the project the model got trained on. Panels across the y-axis indicate the subject projects the corresponding model was tested on. Note that only a single run of experiments is required to derive precision, recall, and f-measure using the three metrics sets in this setup; hence no statistical tests were performed to compare the results from using the different metrics sets.

The cross project results show the same trend as the results derived from the repeated holdout setup. Change genealogy and complexity churn classification models show very similar results. Also, the trend that classifiers based on $C_{\Delta}Ms$ show slight advantages is preserved when switching to the cross-project setup.

☛ *Bug fix classification models trained on change genealogy and complexity metrics can be used as cross-project classifiers with comparable classification accuracy.*

☛ *On a combined data set, there is no statistically significant difference between classification models based on either change genealogy or complexity metrics.*

☛ *Using complexity metrics may be best given that the time to collect the complexity metrics is substantially lower than building and analyzing change genealogy graphs.*

The good classification performances of both classifiers on the cross-project experiment suggests a common subset of driving factors indicating the difference between bug fixes and feature implementing change sets. The result of the repeated holdout experiment over all projects is shown in Figure 6.7 and confirms the strong classification results of the previous experimental setups. Both metric sets yield classification models whose precision lies between 0.66 and 0.7 while the recall values reach from 0.83 to 0.99. A non- parametric statistical test (Kruskal-Wallis) showed that the difference in classification performance is statistically significant ($p < 0.05$).

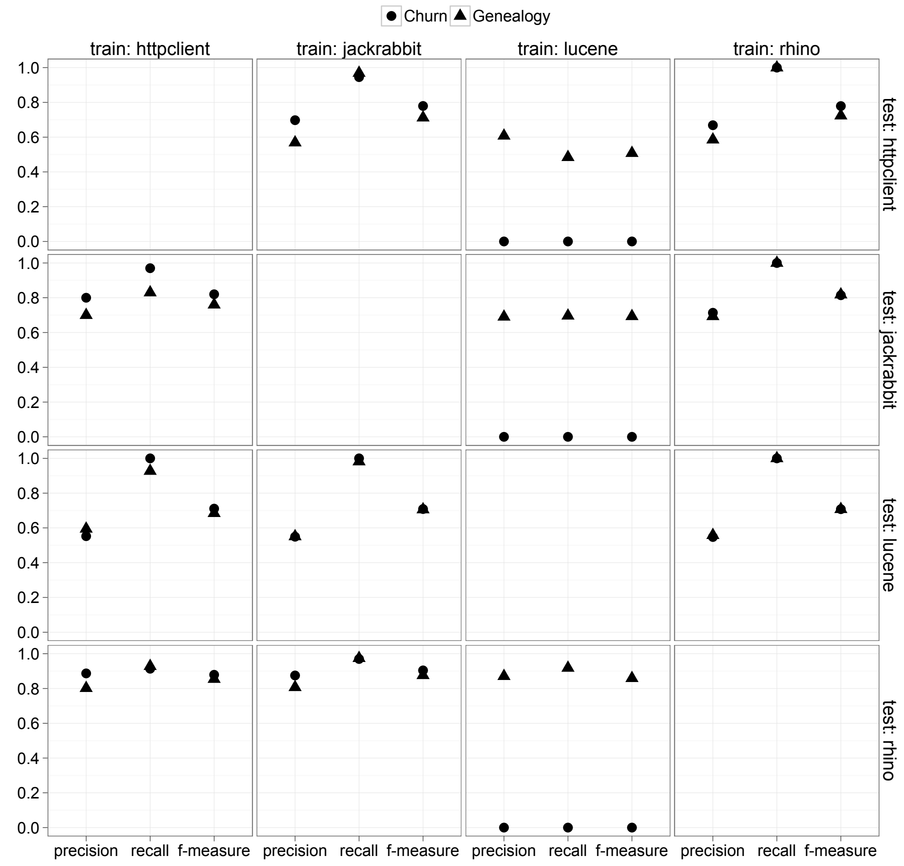


Figure 6.6: Results from the cross-project experiment to separate bug fixing from feature implementing code changes.

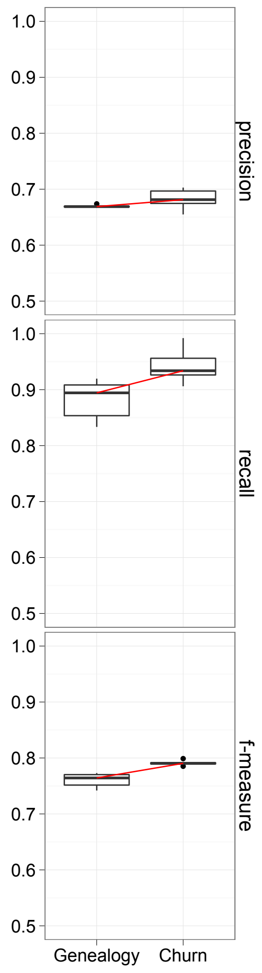


Figure 6.7: Results from the experiment combining all projects to separate bug fixing from feature implementing code changes.

6.4.3 Influential Metrics

The R package *caret* [90] allows computing the importance of individual metrics using the `filterVarImp` function. The function computes a ROC curve by first applying a series of cutoffs for each metric and then computing the sensitivity and specificity for each cutoff point. The importance of the metric is then determined by computing the area under the ROC curve. We used the combined metrics set to compute variable importance for *CGMs* and $C_{\Delta}Ms$ and considered the top-10 most influential metrics for each metrics set for examination.

The most influential *CGMs* are dedicated to the number of applied change operations, code age, the number of change set parents, and network efficiency. Bug fixing change sets seem to apply fewer change operations and change older code while feature implementations are based on newer code fragments. It also seems universal that feature implementing change sets have more structural dependency parents than bug fixing ones.

The most influential complexity difference metrics show that the higher the impact of a change set on cyclomatic complexity of the underlying source code, the higher the chance that the change set is implementing a new feature. Thus, bug fixing change sets show smaller impact on code complexity than feature implementations. Surprisingly, metrics explicitly referring to the size of a change set, such as number of statements, are not among the top ten most influential complexity metrics.

6.5 Predicting Defects (RQ6.2)

This series of experiments is dedicated to research question *RQ6.2*: How do defect prediction models based on *CMs* compare with defect prediction models based on *CMs* or *NMs*? We do not aim to build the best prediction models possible and thus do not make any performance tuning optimizations when training the different prediction models. Our prediction models are trained to classify source code files as containing at least one defect or no defect.

6.5.1 Experimental Setup

To train and test classification models on *CMs* and *NMs*, we can use the originally generated set of metrics as described in Section 6.3.2 and Section 6.3.4.

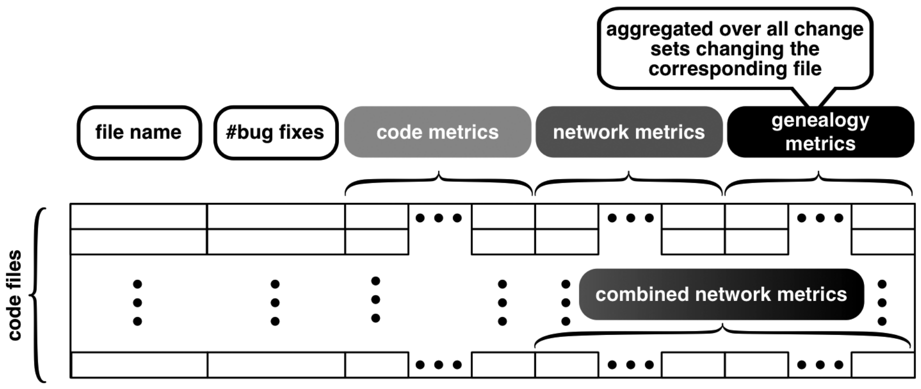


Figure 6.8: Data collection used to predict defects for source files.

The set of *CGMs* cannot be reused without modification. *CGMs* are collected on change set basis and do not refer to source file level. Thus, we have to convert *CGMs* to the source file level. For each source file of the project, we aggregate all *CGMs* values over all change set that modified the corresponding file. We used three different aggregation functions: mean, max, and sum. The resulting data collection is illustrated in Figure 6.8.

Similar to Section 6.4.1 we perform a stratified repeated holdout setup to train and test defect prediction models based on *CMs*, *NMs*, *CGMs*, and a combined network metric set containing both *NMs* and *CGMs*. For each metric set we repeatedly sampled the original data collection (see Figure 6.8) 100 times. For each cross-fold we then trained four series of prediction models (*CMs*, *NMs*, *CGMs*, and combined network metrics set) using six different machine learners (see Table 6.8 on page 123). In total we trained and 9,600 independent prediction models.

6.5.2 Prediction Accuracy

Results from the stratified repeated holdout experimental setup (see Section 6.5.1) are presented in Figure 6.9. Panels across the x-axis in the figure represent the subject projects. The four prediction models were run on 100 stratified random samples on four metric sets: *CMs*, *NMs*, *CGMs*, and a combined set *Combined* combining *NMs* and *CGMs*. For each run we computed precision, recall and F-measure values.

The boxplots in the figure reflect their distribution and for each distribution the best performing model (see Section 6.5.1) is stated under the corresponding boxplot. The black line in the middle of each boxplot indicates the median value of the corresponding distribution. Larger median values indicate better performance on the metric set for the project based on the respective evaluation measure. Note that the red colored horizontal lines connecting the medians across the boxplots do not have any statistical meaning—they have been added to aid visual comparison of the performance of the metrics set. An upward horizontal line between two boxplots indicates that the metrics set on the right performs better than the one of the left and vice versa. Additionally, we performed a non- parametric statistical test (Kruskal-Wallis) to statistically compare the results.

The results shown in Figure 6.9 suggest that network metrics outperform code complexity metrics. Network metric prediction models show better precision and recall values for all four subject projects. Change genealogy models report up to 20% (on average 10%) less false positives (higher recall) when compared to code network metric models. At the same time, recall values for change genealogy models drop slightly in comparison to network metric models. The statistical tests showed that the differences in classification performances are statistically significant ($p < 0.05$).

Models trained on feature vectors combining code dependency and change dependency network metrics show better precision values for HttpClient and Rhino but worse precision values for Lucene when compared to models trained on change genealogy metrics, only. The precision values for Lucene even drop below the precision values of the corresponding network metric models. But interestingly, models trained using the combined metric sets show better recall values for all four projects. For three out of four projects, the recall values are considerable increased (HttpClient, Jackrabbit, Rhino). Additionally, project size seems to have no impact on prediction accuracy.

- ☛ *Code dependency network metrics outperform complexity metrics at predicting defects.*
- ☛ *Models based on change genealogy metrics report less false positives when compared to models trained on either complexity metrics or code dependency network metrics.*
- ☛ *Combining code dependency network metrics and change genealogy metrics yields prediction models with increased recall but decreased precision values.*

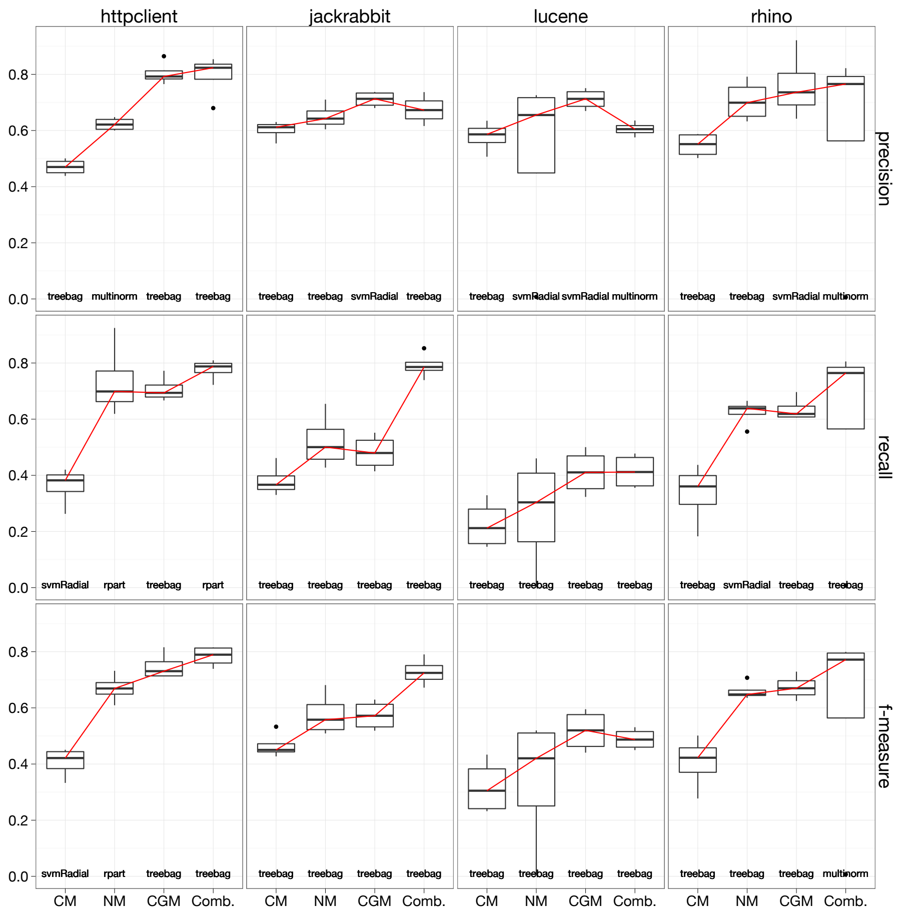


Figure 6.9: Results from the repeated holdout experimental setup. Note that the "Comb." label refers to the combined metric set containing *NMs* and *CGMs*.

6.6 Influential Metrics

We used the same strategy as described in Section 6.4.3 to determine top-10 most influential metrics. For three out of four projects (HttpClient, Jackrabbit, Rhino) seven of the ten most influential metrics are change genealogy metrics. Only for Lucene the top-10 most influential metrics contains no change genealogy metric.

We observed three different patterns with respect to presence and ranking of network and change genealogy metrics. Each of the four top-10 most influential metric sets contained one of the *EffSize* or *Efficiency* metrics as the most important network metrics. For HttpClient, Jackrabbit, and Rhino the top two most influential metrics were change genealogy metrics describing the relation between a change set and its dependencies to earlier applied change sets (outgoing dependencies). The number and type of the dependency parents as well as the time span between the change set and its parents seem to be crucial. The higher the number of parents and the longer the time span between a change set and its parents the higher the probability to add new defects. Thus, code entities changed by many change sets combining multiple older functionality are more likely to be defect prone than others.

☛ *Code entities with a series of changes combining multiple older functionalities are more defect prone than others.*

6.7 Threats to Validity

Empirical studies like this one have threats to validity. We identified three noteworthy threats:

Change Genealogies. Change genealogies model only a dependencies between added and deleted method definitions and method calls. Disregarding change dependencies not modeled by change genealogies might have an impact on change dependency metrics. More precise change dependency models might lead to different change genealogy metric values and thus might change the predictive accuracy of the corresponding classification and prediction models.

Number of bugs. Computing the number of bugs per file is based on heuristics. While we applied the same technique as other contemporary studies do, there is a chance the count of bugs for some files may be an approximation.

Issue reports. We reused a manual classified set of issue reports to determine the purpose of individual change sets. The threats to validity of the original manual classification study (see Chapter 4) also apply to this study.

Non-atomic change sets. Individual change sets might refer to only one issue report but still apply code changes serving multiple other development purposes (e.g. refactorings or code cleanups). Such non-atomic change sets introduce data noise into the change genealogy metric sets and thus might bias the corresponding classification models.

Study subject. The projects investigated might not be representative, threatening the external validity of our findings. Using different subject projects to compare change genealogy, code dependency, and complexity metrics might yield different results.

6.8 Summary

In this chapter, we investigated whether change dependencies modeled by change genealogies are expressive enough to allow automatic bug fix identification. For this purpose, we defined and computed a set of network metrics on change genealogy graphs and used these metrics to build the first automatic bug fix identification model based on change dependencies, only. Comparing this change genealogy change set classification model against a benchmark model based on complexity churn metrics, we showed that both classification models were both able to separate bug fixing from feature adding change sets with a median precision of 70% and a median recall of 85%. Classifiers based on complexity churn metrics seem to have slight advantage when compared to change genealogy based classifiers while change genealogy based classifiers were able to operate on projects for which no complexity churn classifier could be built.

In a second series of experiments, we investigated whether change genealogy can be used for defect prediction purposes. Our assumption was that *crucial code changes* show more dependencies to earlier and later code changes than other code changes and that such crucial code changes are more likely to introduce new defects into source code. We used change genealogy metrics to train classification models separating defect prone source files from non-defect prone source files. The results of this experiment show that change genealogy metrics aggregated to source file level perform well when used as defect prediction feature vectors and report less false positives than models based on the original network metrics. Identifying the most influential metrics using the combined metric set containing network and change genealogy metrics unveiled that code entities applied in order to apply code changes combining multiple older functionalities tend to be more defect prone than others.

The data sets used within these experiments are made public available can be downloaded from the website at:

`http://www.softevo.org/change_genealogies/`

Chapter 7

Predicting Long-Term Cause Effect Chains

Parts of the contents of this chapter have been published in Herzig and Zeller [71].

7.1 Introduction

Software and its reliability is a product of its history, which can be characterized as a sequence of changes. By mining recorded changes, one can identify frequently changing components—an important factor in predicting the risk of defects. And one can discover sets of components that changed frequently together, revealing couplings that are inaccessible to program analysis.

Research studies in analyzing software history have been mostly constrained to either *space* or *time*. Being constrained to *space* means that one examines the evolution of single components, aggregating features over time. Being constrained to *time* means that one examines which components were changed at a single moment in time, extracting co-changes from the resulting transactions. Change genealogies model dependencies between individual code changes telling how changes influence and cause each other. Thus, change genealogies allow us to reason over *multiple* components at *multiple* points in time.

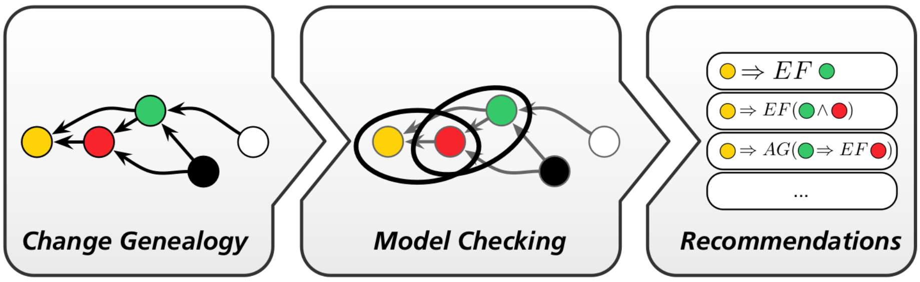


Figure 7.1: Model checking change genealogies to extract frequent rules that describe temporal key features of the underlying software process.

By mining and *model checking* change genealogies, we obtain frequent *temporal patterns* that encode key features of the software process that span both space and time: "Whenever class *C* is changed, its test case is later updated as well." or "Whenever Tom changes his code, Anna has to respond on Tom's change". Once mined, such patterns can be used as building blocks for a software process model; they also can be validated and enforced automatically in further development.

In this chapter, we make again use of the change set change genealogy layer modeling dependencies between all change sets ever applied in a software projects history. We can use change genealogies to *mine* for frequent patterns—in particular *temporal* patterns that span space and time. For this purpose, we make use of the vertex annotations of a change genealogy associating each change set with *features* such as the file or package being modified. We then use *model checking* to determine frequent rules. These rules are expressed in computation tree logic (CTL); a rule such as "$\mathit{file}_1 \Rightarrow \mathsf{EF}\ \mathit{file}_2$" means that whenever file_1 is changed, eventually, file_2 is changed as well. Additionally, the rule implies that the change to file_2 is structural dependent on the change applied to file_1. Such *long-term coupling* based on change set dependencies is not detected by current mining approaches [50, 77, 127, 146, 154] due to their restriction in space and time. Canfora et al. [29] used a sliding window approach to detect change couplings occurring within certain time frames across multiple change sets. But their rules might cover frequent occurring independent activities as their sliding window approach is not based on any change dependency information.

Mined temporal rules can become part of the formal software process model; generally speaking, they encode actions that typically follow immediately or after some period of time. Of course, they can also be validated and enforced automatically: Should file_1 be changed without its test case file_2 following suit before release, chances are that file_1 is not properly tested. When file_1 is changed, we can give a *recommendation* reminding developers that file_2 must eventually be examined—or changed—as well. These recommendations are quite accurate: In an evaluation of four open source histories, GENEVA[1] would recommend pending activities with a precision of 60–72%.

The remainder of this chapter is organized as follows. We first describe how to extract temporal rules (Section 7.2) and how to mine these rules (Section 7.3) using change genealogies. We then evaluate the accuracy and usefulness of the mined rules both qualitatively (Section 7.4) and quantitatively (Section 7.5). We then close with threads to validity in Section 7.6 and conclusion and consequences in Section 7.7.

7.2 Long-Term Couplings

To demonstrate the expressive power of change genealogies and to give a practical working example how to use such change genealogies, we implemented a tool that predicts long-term cause effect chains based on change genealogies. We build on the concept of *change couplings*, introduced by Gall et al. [50] and later improved by Zimmermann et al. [154] and Canfora et al. [29]: If two artifacts are coupled by frequent common changes, we can use this coupling to predict related changes. Such couplings are usually undetectable by program analysis [47].

We consider artifacts frequently or exclusively changed together (within the same change set) to be strongly coupled. Using change couplings, it is possible to detect incomplete code changes, to give recommendations for further changes or to raise awareness that code changes might trigger a number of response changes raising the instability of a software project. The concept of change couplings has an important deficiency, though: It relies on the assumption that coupled artifacts get changed frequently (or always) together within the same change set or at least in small, static time windows after each other. Often, artifacts that are frequently changed across multiple transactions by different authors get disregarded. As an example, consider a `login` function defined in project P_1 and a `service` class defined in project P_2. Changing the `login` function by throwing a new runtime exception requires the developer of class `service` to respond. Both files are maintained by different developers and thus would never occur within one change set. Further, the dependency would not be detected by

[1]GENEVA = GENealogy Extraction from Version Archives.

simply compiling all projects. Still, these files should be considered as tightly coupled. Tools like eROSE [154] do not detect these couplings because they analyze each change set independently. Additionally, approaches like Canfora et al. [29] disregard structural information. Files changed frequently short after each other might not be structurally dependent but got changed due to an iterative development process. To detect cross-transaction change couplings, we need structural dependency information. Considering the temporal order of transactions only does not suffice.

7.2.1 Computational Tree Logic on Genealogies

Change genealogies model dependencies between changes and can be used to extend the concept of change couplings. *Long-term change couplings* are change couplings spanning across multiple software revisions. In terms of change genealogies, we search for software artifact A_1 and A_2 such that whenever A_1 got changed, there exists a genealogy path from A_1 to A_2. The existence of such a path would imply that both changes structurally depend on each other. Each long-term coupling can be seen as a *cause-effect chain:* a developer changes her code and other developers respond to her change; the responding change can again cause other developers to respond to it.

To express long-term couplings, we use *computational tree logic*. CTL is the natural temporal logic interpreted over branching time structures introducing path qualifiers. The above example could be expressed as: $A_1 \Rightarrow \mathsf{EF}\, A_2$—read as "$A_1$ imply exists finally A_2", and means that on every path starting at a change set changing A_1, there is at least one path going through a change genealogy vertex corresponding to a change set changing A_2.

Using CTL to express long-term change couplings allows us to use *formal verification techniques* to *model check* long-term change couplings on change genealogies. While being able to express complex long-term dependency relations in logical formulas, we can also verify those relations automatically on any change genealogy at any time.

CTL is a powerful language with which you can express very complicated temporal properties on a change genealogy graph. Figure 7.2 contains an introduction to model checking and CTL that is taken from Wasylkowski and Zeller [140]. Our goal is to find many plausible CTL formulas that might express long-term cause effect chains. To do so, we use *predefined CTL templates* [61] that are suitable to express long-term cause effect chains. Our CTL formulas make use of only three CTL operators: EF (exists

In general, temporal logic model checking [35] can be used to verify that a system satisfies a specification given as temporal logic formula. The system is typically modeled as a Kripke structure—a finite state automaton. CTL (computational tree logic) [34] is a temporal logic language used to express specifications. Let AP be a set of atomic propositions. A Kripke structure over AP is a tuple $M = (S, I, R, L)$, where S is a finite set of states, $I \subseteq S$ is the set of initial states, $R \subseteq S \times S$ is a left-total transition relation, and $L : S \rightarrow 2^{AP}$ is a labeling function. Atomic propositions are used to describe the state the system is in, and the Kripke structure represents transitions between the states of the system. Because R is a left-total relation, all the behaviors of the system are infinite. CTL is a temporal logic used to predicate over the behaviors represented by the Kripke structure. It is defined over the same set AP of atomic propositions that the Kripke structure uses:

1. *true* and *false* are CTL formulas
2. Every atomic proposition $p \in AP$ is a CTL formula
3. If f_1 and f_2 are CTL formulas, then so are $\neg f_1$, $f_1 \vee f_2$,$f_1 \wedge f_2$, $f_1 \Rightarrow f_2$, and $f_1 \Leftrightarrow f_2$.
4. If f_1 and f_2 are CTL formulas, then so are $\mathsf{AX}f_1$, $\mathsf{EX}f_1$, $\mathsf{AF}f_1$, $\mathsf{EF}f_1$, $\mathsf{AG}f_1$, $\mathsf{EG}f_1$, $\mathsf{A}[f_1 \bigcup f_2]$, $\mathsf{E}[f_1 \bigcup f_2]$.

A means "for all paths", and E means "there exists a path". X stands for "next", F stands for "finally", G stands for "globally", and U stands for "until". The intuitive meaning of some CTL formulas is as follows: $\mathsf{AX}f_1$ means that "for each state $s_0 \in I$, for all (A) paths starting in s_0, f_1 holds in the next (X) state". $\mathsf{EF}f_1$ means that "for each state $s_0 \in I$, there exists (E) a path, where f_1 holds somewhere along (F) this path". $\mathsf{AG}f_1$ means that "for each state $s_0 \in I$, for all (A) paths starting in s_0, f_1 holds in all states along (G) the path". $\mathsf{A}[f_1 \bigcup f_2]$ means that "for each state $s_0 \in I$, for all (A) paths starting in s_0, f_1 holds until (U) f_2 holds" (i.e., f_2 must hold somewhere along the path, and until then, f_1 must always hold). An atomic proposition p holds in a given state iff this state is labeled with p. Model checking a given CTL formula f against a given Kripke structure $M = (S, I, R, L)$ is equivalent to asking if f holds for each $s_0 \in I$. If it does, f is said to be true for M; if it does not, f is said to be false for M.

Figure 7.2: CTL and model checking in a nutshell. Taken from Wasylkowski and Zeller [140].

finally), EX (exists next), and AG (all globally). We also limit the number of involved source code artifacts to three:

***Templ*$_1$:** $A_1 \Rightarrow \mathsf{EF}\, A_2$: Changing A_1 cause at least one dependent change on A_2.

***Templ*$_2$:** $A_1 \Rightarrow \mathsf{EF}(A_2 \wedge A_3)$: Changing the artifact A_1 causes dependent changes in A_2 and A_3 within the same change set.

***Templ*$_3$:** $A_1 \Rightarrow \mathsf{EF}\, A_2 \wedge \mathsf{EF}\, A_3$: Changing A_1 causes dependent changes in A_2 and A_3 but not necessarily in the same change set.

***Templ*$_4$:** $A_1 \Rightarrow \mathsf{AG}(A_2 \Rightarrow \mathsf{EF}\, A_3)$: Changing A_1 and later changing A_2 causes a change in A_3. All later changes depend on the initial change of A_1.

We do not claim these formulas as complete. In fact, you can choose any CTL formula.

While CTL formulas are defined on software artifacts a change genealogy graph expresses dependencies between change sets. To bridge this gap, we use the change set change genealogy layer. Replacing the template's artifact placeholders A_1,A_2, and A_3 with the corresponding changed source files, we can express temporal source file dependencies over change genealogy paths.

To illustrate this procedure, lets consider the change genealogy from Figure 3.6 on page 41 using CTL template *Templ*$_1$. We choose a path that matches the temporal logic of *Templ*$_1$: $\{CS_1, CS_3, CS_5\}$. Within *Templ*$_1$ we replace the variable A_1 with the file names changed within CS_1: $File_1$, $File_2$ and the variable A_2 with those file names changed within CS_5 : $File_4$. The resulting formulas are: $File_1 \Rightarrow \mathsf{EF}\, File_4$ and $File_2 \Rightarrow \mathsf{EF}\, File_4$. Later (Section 7.2.3) we will see that not all possible template instances have to be generated.

For clarification: consider the change genealogy shown in Figure 3.6 and replace A_1, A_2, and A_3 by $File_1$, $File_2$, and $File_3$ in the four templates above. The formula derived from *Templ*$_1$ holds when model checked, because all path start in CS_1 and there exists a path covering a change in $File_2$: CS_3 (same holds for *Templ*$_3$). The formula from *Templ*$_2$ does not hold. There exists no vertex that changes $File_2$ and $File_3$ together. Finally, the formula from *Templ*$_4$ does not hold, because the path CS_1,CS_3,CS_4 does not change $File_3$.

☛ *Long-term couplings detected by GENEVA imply structural dependencies between coupled artifacts.*

7.2.2 Limiting the Temporal Scope

In the previous section, we discussed how to generate CTL formulas using CTL templates. But in most projects all such dependencies will become eventually true. Even though each temporal path expressed by a long-term coupling rule is based on structural code dependencies, we want to limit the temporal scope in which such long-term coupling rules must be valid. The longer the time between two dependent nodes, the lower the probability that the later applied change was caused by the earlier one. Therefore, we limit the number of days between the initial change and the depending changes. Doing so, we can ensure that there is a maximal time window (*maxdays*) in which a CTL formula must be valid.

For this purpose, we use a sliding window approach to generate multiple *genealogy sub graphs* from the original change genealogy graph. For each vertex u of the original change genealogy graph $G(V, E)$ we generate a corresponding sub graph $G'(V', E')$ such that:

$V' = \{v \in V \mid t(u, v) \leq max\_days \ \wedge \ path(u, v)\} \cup \{u\}$
where $t(u, v)$ equals the number of days between the commit dates of u and v. $path(u, v)$ is true if and only if there exists a path from u to v in G.

$E' = \{e(v_1, v_2) \in E \mid v_1 \in V' \ \wedge \ v_2 \in V'\}$

Figure 7.3 shows an example genealogy sub graphs extracted from our initial genealogy in Figure 3.6 on page 41 ($maxdays = 1$). Each genealogy sub graph is a connected graph having u (here CS_1) as a root. The number of *generated* genealogy sub graphs equals the number of vertices in the original genealogy graph G. Graphs with a depth of one can be ignored.

Model checking the CTL formulas on the genealogy sub graphs ensures a time window in which these formulas have to be valid. Additionally, we automatically ignore changes with no outgoing dependency. This reduces the model checking space and makes model checking all these formulas feasible.

7.2.3 Model Checking Genealogies

At this point, we have introduced the concept of change genealogies and explained the concept of long-term change couplings described as CTL formulas. To *derive* valid

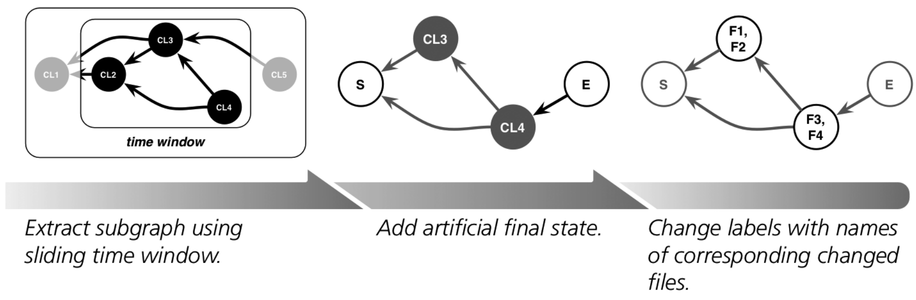

CL1
CL2
CL3
CL4
CL5
time window
S
E
CL3
CL4
F1, F2
F3, F4
Extract subgraph using sliding time window.
Add artificial final state.
Change labels with names of corresponding changed files.

Figure 7.3: Extracting change genealogy sub graphs from change genealogy graph using sliding time window and converting change genealogy sub graphs to Kripke structure.

long-term couplings, we *model check* which CTL formulas hold on the genealogy sub graphs. But before, we have to transform each genealogy sub graph into a *Kripke structure*—a nondeterministic finite state machine whose nodes represent reachable states with transition edges between them.

Our genealogy subgraphs discussed in Section 7.2.2 are connected, directed, and acyclic. They can be interpreted as a nondeterministic finite state machine and therefore already are Kripke structures. It remains to add a new artificial final state (the graph will be left-total) and replacing the original node labels by those filenames changed by the corresponding change set (using the vertex annotations). This way, our Kripke structures express temporal dependencies between changed files instead of change sets.

The resulting Kripke structures can be used to model check our CTL formulas. Formulas evaluating to true on at least one Kripke structure are worth further investigation since they represent potential long-term coupling rules.

7.3 Long-Term Coupling Rules

In the previous section, we generated valid CTL formulas and prepared our data model to allow automatic CTL validation. But which formulas describe *frequent change patterns*? Which formulas occurred rarely or only once? To mark important rules and to determine the strength of a rule we rank rules by their *support* and *confidence* measures [154]:

Support. We measure the number of Kripke structures on which the CTL formula f was evaluated to be true as **support(f)**. For the example genealogy graph shown in Figure 3.6, the formula $File_1 \Rightarrow \mathsf{EF}\, File_3$ has a support of two. $File_1$ was changed in change sets CS_1 and CS_3. The change sets CS_2 and CS_4 change $File_3$ and depend on either CS_1 or CS_3.

Confidence. To measure the strength of the consequence expressed in formula f we calculate **confidence(f)** as the fraction of the formula's support divided by the number of times the premises has occurred. For the example genealogy graph shown in Figure 3.6, the formula $File_1 \Rightarrow \mathsf{EF}\, File_3$ has a confidence value of one. The support value of the formula is two (see above) and $File_1$ got changed twice.

Rules are primarily ranked by confidence. Rules with equal confidence are ranked by support.

7.3.1 Rules as Recommendations

The purpose of change couplings is to be used as recommendations. Whenever a programmer commits a change, a recommendation tool suggests further changes based on rules extracted from earlier code changes. But change couplings can also be used in retro perspective to decide whether an applied refactoring worked (the refactoring should uncouple entities) or to simply identify frequent occurring change patterns that can be used for further software development process analyses. Long-term change rules express frequent change rules that span multiple change sets. Thus, each recommendation based on long-term change rules does not suggest code changes to be made within the same change set but might indicate future development activities; further changes to be applied by a different developer within a time window of *maxdays* number of days.

The computation of long-term coupling rules is already described in Section 7.2. To compute recommendations for a given change set CS_i, we have to execute the following steps:

1. Generate the change genealogy graph until CL_{i-1} and generate all valid CTL formulas within the current genealogy sub graph. That is, extract all changed files from all change sets of the genealogy sub graph and create all possible CTL formulas if not in the CTL cache. Each CTL formula is stored as *long-term coupling rule* together with the up-to-date support and confidence values.

2. Select all long-term coupling rules with implication premises that correspond to files changed within CL_i.

3. Rank the selected rules by confidence and support.

The computation of change genealogies and CTL rules, model checking them and computing support and confidence values on the fly takes time. Generating all long-term coupling rules for a mid-size project spanning a history of 15,000 transactions take several hours—depending on the average number of files changed by transactions. (Change genealogies must not be regenerated for change set but can be extended by single or multiple change sets.) Still, to improve efficiency of GENEVA, we used the following optimizations:

Ignoring large change sets. Change sets that touch many files are suspicious because most of them either combine multiple changes that should be separated, refer to refactoring or documentation updates. In general, such vertices have a large in-degree and out-degree causing millions of CTL formulas generated with higher probabilities to cause false positives.

We determine the median number of files changed by earlier change sets. Change genealogy vertices for which the number of changed files is larger than the 3/4-quantile of the change size distribution are ignored.

Ignoring rarely changed files. In a project history, there are many code entities that have a very limited life span. Other entities are rarely updated, if ever. To minimize the number of relevant CTL formulas and to ease the memory consumption, we ignored all source files that were changed only once.

Ignoring deleted files. We drop rules that would contain source files deleted within the transaction as implication. Whenever a source code entity gets deleted, we remove all CTL formulas that have the deleted artifacts as implication premise.

Optimized, generating long-term coupling recommendations for single change set is fast and takes about one second.[2]

[2]All times were measured on a Linux Server using a single Intel Xeon(X5570) processor (2.93GHz) and 8GB RAM.

7.3.2 Additional Change Properties

There might be cause-effect chains that occur under certain *circumstances* only. To capture cause-effect chains that are bound to certain development activities, we have to combine long-term coupling rules with *change properties* such as size, author, or purpose of a change.

We implemented two such properties: *fixes* and *big changes*. Analyzing commit messages similar to Zimmermann et al. [153], we can determine if the applied changes were made to fix a bug. Change sets changing more than 20% of a file's content are classified as *big changes* with respect to the individual file. Formulas with general low confidence might have high confidence when considering fixing change sets or big changes only. An example of such *conditional* long-term coupling rule is given in Section 7.4.

7.3.3 Inner-Transaction Rules

So far, GENEVA extracts long-term coupling rules occurring across multiple change sets only. Previous work [154, 29] also reported rules that occur within the very same change set. For comparison purposes, we added an option adding *inner-transaction rules* to adjust formulas support and confidence values.

Our main purpose of this study is highlighting the contribution of long-term couplings. Thus, we obtained all results with disabled inner-transaction rules. In Section 7.5.7, however, we will see that inner-transaction rules increase the number of recommendations without sacrificing the precision.

7.4 Example Long-Term Coupling Rules

GENEVA uncovers long-term change couplings. Unlike couplings within change sets, long-term change couplings might not be obvious nor directly understandable by looking at the code in one version—simply because they span a longer period of time. To project outsiders, many long-term change rules may come as surprises (which may actually add to their value). Below, we give three basic examples of long-term coupling rules from the JRuby project in CTL style.

Long-term couplings in JRuby. The JRuby project is based on a large compiler infrastructure defining many compiler interfaces and implementations for these interfaces. One such interface is called `VariableCompiler`. It changed 18 times between 2001 and 2010. Out of these 18 changes, 16 caused a change in `StandardASMCompiler` within 8 days. Each time, both change sets depended indirectly on each other. Considering the complete JRuby project history, the long-term coupling rule has a support value of 16 and a confidence of 0.77.

$$\texttt{VariableCompiler} \Rightarrow \mathsf{EF}\,\texttt{StandardASMCompiler}$$

Although, both files were never changed together, the same developer maintains both files. Both classes even call each other indirectly: `StandardASMCompiler` uses `BodyCompiler` as interface; its implementation `BaseBodyCompiler` then references `VariableCompiler`. The coupling although frequent, can only be detected over time—by using an approach like GENEVA.

Test suite changes. Adding functionality to classes often requires new test cases to be added. Such dependencies occur frequently within the same change set. But there are also cases in which changed test cases unveil problems in classes that might not be directly under test. These cases often span multiple change sets since the newly discovered issues cannot be checked ad hoc:

$$\texttt{MainTestSuite} \Rightarrow \mathsf{EF}\,\texttt{RubyObject}$$

Nine changes to the `MainTestSuite` made other developers change the class `RubyObject`. The obvious assumption is that the main test suite unveiled new problems in `RubyObject`; the long-term coupling rule connecting both files has a confidence of 0.65.

For JRuby we found 8 rules having a test case as premises with an average confidence of 0.59 and an average support value of 6.25. Note that most of these case could not be detected by compiling the project(s) due to the fact that most test errors are runtime issues.

Fixes vs. changes. Some change couplings occur only under certain circumstances. The following rule has an overall confidence value below 0.5:

$$\texttt{RubyIO} \Rightarrow (\mathsf{EF}\,\texttt{RubyStructure} \wedge \mathsf{EF}\,\texttt{Visibility})$$

However, if the changes applied to `RubyIO` are bug fixes, the rule has a confidence of 0.8. In other words, fixes to `RubyIO` imply other changes, while regular changes do not.

For JRUBY we found 31 long-term coupling rules that only occur frequently when fixing an artifact. The average confidence of these rules lies at 0.63 with an average support value of 5.4.

All these rules span both space and time, and reveal long-term couplings that GENEVA is the first approach ever to uncover. Deviations from these rules are likely candidates for missing activities and hence problems.

☛ *GENEVA can use additional change properties as coupling conditions to detect change coupling rules that cannot be detected by comparable tools.*

7.5 Quantitative Evaluation

After having explored some of the patterns manually, we wanted to know how many such rules exist and how useful these patterns are in practice. The fact that long-term coupling rules are represented in CTL formulas allows *automatic pattern validation* on other change genealogies. Therefore, the accuracy of this validation is our evaluation target: How reliable are these rules and do patterns really occur frequent enough to allow recommendations?

7.5.1 Evaluation Subjects

For our quantitative evaluation, we chose four open source projects that had more than two years of project history and were under constant development by more than twenty developers (see Table 7.1). The project histories contain seven to twelve years of active development, more than 1,300 project revisions and more than 1,000 changed source files. In our experiments we bound out analysis to the main development branches. The number of those files causing long-term couplings varies from project to project. For three out of the four projects GENEVA found over 200 long-term coupling rules with confidence above 0.5 and for two projects nearly 100 long-term couplings with confidence above 0.7.

Table 7.1: Evaluation subjects. '#Files' is the number of files changed within the project history. The rows 'Number LTC ≥ 0.5' and 'Number LTC ≥ 0.7' show the number of source files for which long-term couplings with support values ≥ 3 and a confidence ≥ 0.5 and ≥ 0.7 respectively exist.

	ArgoUML	Jaxen	JRuby	Xstream
History length	12 years	9 years	9 years	7 years
Number of authors	50	20	66	11
Number of source files	16,658	9,831	15,029	1,188
Change genealogy details				
Number of vertices	9,426	618	7,848	808
Number of edges	45.130	2,297	47,940	2,709
Mean vertex out degree	4.8	3.7	6.1	3.4
Median youngest child gap	16	9	8	4
Number LTC ≥ 0.5	243	28	232	231
Number LTC ≥ 0.7	94	10	99	19

7.5.2 Exploring Change Genealogy

Table 7.1 shows details of the change genealogy graphs extracted from the four project histories. The number of vertices equals the number of transactions in the main development branch `trunk` changing at least one Java source file. GENEVA cannot determine dependencies between non-Java files and thus ignores these files.

For each project, we had to determine an appropriate window size *maxdays*. Choosing *maxdays* too small will disregard many potential long-term couplings, but choosing it too large will spoil the results by adding a lot of noise. As a first approximation, we therefore used the change genealogy graphs to compute the *median number of days between two dependent transactions* to determine a reasonable value for the window size to be used. The values in row "Median youngest child gap" of Table 7.1 show that the median time gaps between vertices and youngest child vary between four and sixteen days.

7.5.3 Predicting Long-Term Couplings

To evaluate whether long-term coupling rules are precise enough to be used for recommendations, we build a long-term coupling pattern prediction model. GENEVA ranks recommendations by their confidence and support values. The top ranked three recommendations are then used as prediction result. The first 10% of each project history define the training period allowing GENEVA to learn common long-term change patterns and rules. For the remaining 90% of transactions, we used GENEVA to predict the top three ranked long-term coupling rules:

1. Let CL be the change set to predict rules for. Further, let CF_{CL} be the set of files changed by CL.
2. Remove all files $f \in CF_{CL}$ from CF_{CL} that got removed by applying CL: $CF'_{CL} = CF_{CL} \setminus \{f \in CF_{CL} | f$ got deleted by $CL\}$. Remove all rules that have f as implication premise.
3. Take all CTL rules seen in the past that have file $c \in CF_{CL}$ as implication premise and store as prediction candidates PC_{CL}.
4. Remove all entries from PC_{CL} that have a confidence lower than 0.5 or a support value lower than 3: $P_{CL} = \{pc \in PC_{CL} | conf(pc) \geq 0.5 \ \wedge \ support(pc) \geq 3\}$.
5. Sort P_{CL} by confidence. Rank entities with equal confidence using their support value. Remove all but the top three entities from P_{CL}.
6. Let G_{CL} be the change genealogy sub graph for the change set CL using *maxdays* (see Table 7.1). Let $F_{G_{CL}}$ be the set of CTL formulas that evaluate to true on G_{CL}.
7. Update support and confidence values of all know formulas and add new rules.

7.5.4 Benchmark Model

To illustrate the usefulness of this approach, we compare the prediction measurements with a very basic benchmark model that constantly predicts the top three most changed files, at the prediction point in time. In Table 7.2, benchmark values are stated behind the GENEVA result in brackets.

We explicitly avoided using earlier change coupling approaches [154, 29] as benchmark models. The reason is that these earlier approaches do not consider structural dependencies and do not aim to detect and predict long-term cause effect chains but rather incomplete code changes. Using no structural information when detecting change couplings can produce report rules of frequently occurring but independent activities. Such activities might raise support and confidence values but do not refer to cause-effect chains. In most cases we were also unable to reproduce the results of earlier studies. Zimmermann et al. [154] reported an average precision of 0.5 for detecting removed change operations within change sets. Canfora et al. [29] reported that association rules across time windows had precision values between 0.5 and 1.0, depending on the project. These vales can be used to compare our approach with earlier studies. But keep in mind that the approach presented in this paper is to the only one that considers structural code dependencies to discover and predict long-term change couplings.

7.5.5 Precision of Recommendations

As performance measurement for our prediction models, we compute their *precision*. A standard metric for the fidelity of classifications, the precision determines the fraction of correctly predicted long-term change coupling rules:

$$precision = \frac{\#true\ positives}{\#true\ positives + \#false\ positives}$$

Table 7.2 shows the prediction results of the described prediction process. The precision of the prediction model lies between 60 and 72 percent—thus, roughly two out of three recommendations correctly predict a future code change that will depend on the current code change within the time frame of *maxdays*. This precision is on par with systems like Zimmermann et al. [154] but clearly outperforms the precision of the benchmark model. Given that the recommendations are based on structural change dependencies and span space *and* time and thus face a far greater challenge, this is a very satisfying result.

Precision is usually accompanied by *recall*, a measure of completeness of our predictions. In our setting, this would mean to evaluate how much of a system's future evolution (as expressed by future long-term couplings) is predictable from its past history. Since the future is determined by so many factors that are completely outside of the domain of our research (and far out of the capabilities of any research), the measure of recall makes little sense in our context.

☛ *In our evaluation, two out of three recommendations by GENEVA correctly predict the next structural dependent development activity.*

Table 7.2: Prediction results for long-term couplings (benchmark results in brackets).

	ARGOUML	JAXEN	JRUBY	XSTREAM
	GENEVA			
Precision	0.60 (0.31)	0.70 (0.28)	0.72 (0.58)	0.63 (0.28)
Mean rank of highest hit	1.8 (2.0)	1.8 (2.4)	1.9 (2.1)	1.8 (2.1)
Changes recommended	9.2%	9.3%	32.6%	20.7%
Changes without false recommendation	52.3%	68.8%	58.0%	54.2%
	GENEVA with inner-transaction rules			
Precision	0.66	0.59	0.71	0.67
Mean rank of highest hit	2.0	2.0	2.0	2.1
Changes recommended	21.5%	17.2%	43.8%	37.1%
Changes without false recommendation	48.0%	47.8%	49.1%	43.8%

7.5.6 Efficiency of recommendations

In GENEVA's recommendations, the average rank position of the highest ranked true positive (row "Mean rank of highest hit" in Table 7.2) lies between one and two. This means that in case GENEVA gives a recommendation, the first or second recommendation is a hit. Thus, choosing a recommendation list length of more than three would still deliver valid long-term coupling rules at the top of the recommendation list.

☛ *GENEVA's recommendations are efficient, placing the correct activity in the top two positions of the ranked list.*

Rows three and seven in Table 7.2 ("Changes recommended") shows the percentage of change sets that triggered GENEVA to recommend further changes. Only between nine and thirty percent of all change sets actually trigger a recommendation. The percentage of recommendations that contain no false positives (rows four and eight) is high. More than 50% of recommendations contain no false positive.

☛ *More than 50% of GENEVA's recommendations contain no false positives.*

7.5.7 Adding inner-transaction rules

The lower half of Table 7.2 shows prediction results for the enhanced GENEVA tool integrating inner-transaction coupling rules (Section 7.3.3). With inner-transaction rules, the prediction precision for the projects ArgoUML and Xstream slightly improved, while the precision for Jaxen and JRuby decreased. Overall, though, it seems that the integration of inner-transaction change patterns does not add many coupling rules not known by GENEVA before. The average rank of the highest true recommendation slightly drops but remains stable across all projects. Surprisingly, the number of vertices for which GENEVA gives recommendations increases drastically. Together with a stable to slightly improved precision, we can conclude that adding inner-transaction rules to GENEVA improves the overall results. This also implies that both sets, inner-transactional and long-term couplings, are not subsets of each other. Increasing the number of recommendations without decreasing precision implies that GENEVA added rules that could not be detected within single transactions and vice versa.

☛ *Adding inner-transaction rules to GENEVA increases the number of recommendations without sacrificing precision.*

7.6 Threads to Validity

External validity. We only examined the histories of four open-source projects. We cannot claim their development process or project history is representative for other projects. However, we expect projects with a tighter process control to result in more process rules and increased accuracy.

Internal validity. Our approach of modeling dependencies between changes by methods is kept simple on purpose, and is neither necessarily sound nor necessarily complete. As the problem is generally undecidable, a certain amount of imprecision cannot be avoided. Concepts like the coupling between transactions (rather than atomic changes) or the use of time windows may also reduce precision while improving efficiency.

Construct validity. Our evaluation used a standard approach: Learning from the past and checking whether our findings still hold in the future; no manual interpretation was involved that could threaten our findings.

7.7 Summary

In software development, activities are spread across space and time—and yet depend on each other. Change genealogies capture these dependencies as long-term couplings between changes and affected entities. Our prototype GENEVA is able to detect such long-term couplings as *temporal rules* that capture key features of the underlying software process. Using computational tree logic to express these temporal rules and using model checking to determine the validity of these temporal rules on change genealogies, GENEVA predicts code changes that will be applied in future with a precision around 70% (Table 7.2), which is considerably higher than predicting the most frequently changed files. All recommendations and rules learned are based on structural dependencies and, thus, do not contain temporal rules expressing dependencies between frequent occurring but independent development activities. Being able to predict across space and time shows the potential of change genealogies in predicting software features.

Chapter 8

Conclusions and Future Work

Constant changes in requirements induce constant code changes that are spread across space and time——and yet depend on each other. Understanding the complexity of software development activities and being able to track back on which decisions code changes are based, provides fundamental information that determines the quality of a change. Change genealogies capture these dependencies between changes and affected artifacts and can be used to estimate the purpose, quality, and long-term impact of code changes.

But the results of this thesis also show that mining software archives cannot be seen as full automation of empirical software engineering. There exist widespread issues concerning tangled changes and the separation of bug and non-bugs. Trusting raw repository data without checking for noise and bias can severely impact the accuracy and thus the usefulness of any mining tool and study, which leverages such data.

This thesis makes the following contributions to improve mining software repositories:

Change genealogies We introduced *change genealogies*, a graph structure that models dependencies between code changes. Using change genealogies it is possible to track back which decisions caused which code changes providing fundamental information that determines the quality of a change. Change genealogies model dependencies between code changes applied at different times and affecting different code artifacts using structural code dependencies that cannot be detected by standard mining techniques.

Bug report classification Manually inspecting and classifying over 7,000 issue reports from five open source projects we provided evidence of widespread issues regarding the separation between bugs and non-bugs in software archives. These issues can severely impact the accuracy of any tool and study that leverages bug data without performing a qualitative analysis of the input data. The publicly available data sets containing the results of our the manual inspection can be used as ground truth data set for further research and mining model calibration.

Untangling changes Tangled change sets are change sets that combine code changes serving multiple developer maintenance tasks. Tangled change sets are common in version control systems and cannot be avoided. We provides evidence that the fraction of tangled change sets may be substantial, causing a serious threat to empirical findings based on version archives. We proposed an untangling algorithm that helps to reduce the amount of bias within data mining sets, caused by version archive commits, combining changes that were committed due to multiple developer maintenance tasks. Our algorithms showed an average precision between 67% and 93% when untangling any two artificially tangled change sets. The mean precision across all blob sizes is 58% (blob size four) to 80% (blob size two).

Change Genealogy Metrics We transformed the concept of network metrics as proposed by Zimmermann an Nagappan [151] to change genealogies.

Change classification We then used these change genealogy metrics to automatically separate bug fixes from feature adding change sets without using commit message analysis nor bug databases. For this purpose, we trained and tested classification models to separate bug fixing from feature implementing change sets using change genealogy metrics as feature vectors. Overall, classification models based on change genealogy metrics showed a median precision of 0.69 and a median recall of 0.81. We also showed that these change purpose classification models can be used as cross-project classification models.

Defect prediction We also showed that change genealogy metrics can be used to train prediction models to identify defective source files. Our results show that models based on change genealogy metrics report less false positives when compared to models trained on network metrics. An investigation about the most influential change genealogy metrics when classifying defect prone entities unveiled that code entities with a series of changes combining multiple older functionalities are more defect prone than others.

Long-term cause effect chains We introduced GENEVA to predict long-term cause effect chains using CTL to express long-term coupling rules that imply structural dependencies between coupled artifacts. Using these CTL rules and a directed, acyclic version of change genealogies we were able to use the formal method of model checking to extract the set of valid long-term coupling rules. Using additional change genealogy properties as coupling conditions we can detect change couplings that cannot be detected by comparable tools. In the evaluation, two out of three recommendations by GENEVA correctly predict the next structural dependent development activity. More than 50% of GENEVA's recommendations contain no false positive.

8.1 Lessons Learned

The importance of change dependencies. The results presented in this thesis provide evidence that change genealogies are a valuable mining source. Code changes provide historic evidence how source code evolved and gives partial explain the state of a project's source code. Expressing dependencies between code changes across multiple points in time and across multiple components allows to reason about the long-term impact of code changes and their importance in the past development process.

Bug data must be used with care. Bug databases are a frequently used data source when measuring or estimating quality related properties. But our manual classification study presented in Chapter 4 showed that raw bug data sets are "polluted" with misclassified issue reports.

Tangled changes must be untangled. The problem of tangled code changes is well known and in fact, there exist good reasons why developers tend to commit multiple changes together. But current approaches mining version archives tend to ignore these issues. Although, the presented untangling algorithm presented in Chapter 5 might not be precise enough, our approach shows the potential impact of tangled change sets on mining approaches such as defect prediction models.

Importance of human qualitative analysis. Manual inspections and data analysis require a large amount of human efforts involves inspecting thousands of data samples but are most important. Common assumptions on data sets and their quality must be proven before being used. In general, any automated quantitative analysis should always include human qualitative analysis of the data—and of the findings. This is basic quality assurance.

8.2 Future Work

This work improves common mining techniques by introducing a graph like version archive model capturing dependencies between code changes that allows model checking version archives. Additionally, the results of this thesis also provide evidence of major data issues in many mining models, but also provides ground truth data sets and algorithms to reduce the impact of these data noise and bias sources on state-of-the-art mining models. The presented approaches can be improved and extended and the following research questions should be investigated.

It's not a bug. It's a feature.

The manual inspection of over 7,000 issue reports and the high misclassification rates are alarming. With their automated bug report classifier, Antoniol [8] already provided a partial solution to this problem. But we definitely need further studies showing the exact impact on defect prediction and other quality related models.

- **Impact on real defect prediction models.** It will be important to measure and describe the impact of our findings on real defect prediction models. Unless we can verify that defect prediction models are severely impacted, these finding remain primarily. It also remains unclear if current defect prediction model predict defects or changes. Both measures might be highly inter-correlated but are not the same.
- **Impact on other quality related models.** So far, we only showed the impact of these data sets on defect prediction models. But there are many other quality related types of models that might be affected as well.

Untangling Changes

Our results indicate that untangling changes is a surprisingly difficult task, leaving lots of room for future improvements. Nevertheless, we showed that between 6% and 50% of the most defect prone entities are falsely classified as such, due to tangled change sets. Thus, the issue of tangled change sets is serious and needs to be resolved. Our future work will focus on the following topics:

- **Non-essential changes.** So far, we do not identify non-essential changes as proposed by Kawrykow and Robillard [84]. We can use the author's concept to exclude non-essential changes before untangling, putting all non-essential changes into a separate change set partition. We believe that such an integration of both approaches would improve the precision of the untangling algorithm.

- **Recommendations to developers.** In the long run, one could also present untangled change sets as recommendations to users—either at the time they are committed, or at the time they would be manually analyzed. Our current focus, however, is to reduce data noise and the resulting threats to analysis of version archives.

Predicting Long-Term Cause Effect Chains

Besides general improvements to performance and scalability, future work should focus on:

- **Dependencies between changes.** Currently, we are investigating dependencies between *changes* to allow GENEVA to use a much finer-grained dependency graph to increase prediction precision.

- **More features.** Right now, we express temporal patterns over individual files affected by a change. Rules may also include authors ("Whenever Bob changes something, Alice revises it") or metrics ("If cyclomatic complexity exceeds 0.75, a module will be refactored").

- **Graph patterns and metrics.** Besides temporal rules, we can also search for specific *patterns* in the genealogy graph, such as identifying changes that trigger the most future changes, or the changes with the highest long-term impact on quality or maintainability.

Bibliography

[1] git. Available at: http://git-scm.com/.

[2] Jhawk 5 (release 5 version 1.0.1). Available at: http://www.virtualmachinery.com/jhawkprod.htm.

[3] mercurial. Available at: http://mercurial.selenic.com/.

[4] subversion. Available at: http://subversion.apache.org/.

[5] Alam, O., Adams, B., and Hassan, A. E. A study of the time dependence of code changes. In *Proceedings of the 2009 16th Working Conference on Reverse Engineering* (Washington, DC, USA, 2009), WCRE '09, IEEE Computer Society, pp. 21–30.

[6] Albrecht, A. J., and Gaffney, J. E. Software function, source lines of code, and development effort prediction: A software science validation. *IEEE Trans. Softw. Eng. 9*, 6 (Nov. 1983), 639–648.

[7] Amor, J. J., Robles, G., and Gonzalez-Barahona, J. M. Effort estimation by characterizing developer activity. In *Proceedings of the 2006 international workshop on Economics driven software engineering research* (New York, NY, USA, 2006), EDSER '06, ACM, pp. 3–6.

[8] Antoniol, G., Ayari, K., Di Penta, M., Khomh, F., and Guéhéneuc, Y.-G. Is it a bug or an enhancement?: a text-based approach to classify change requests. In *Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds* (New York, NY, USA, 2008), CASCON '08, ACM, pp. 23:304–23:318.

[9] Anvik, J., Hiew, L., and Murphy, G. C. Who should fix this bug? In *Proceedings of the 28th international conference on Software engineering* (New York, NY, USA, 2006), ICSE '06, ACM, pp. 361–370.

[10] Anvik, J., and Murphy, G. C. Reducing the effort of bug report triage: Recommenders for development-oriented decisions. *ACM Trans. Softw. Eng. Methodol. 20*, 3 (Aug. 2011), 10:1–10:35.

[11] Basili, V. R., Briand, L. C., and Melo, W. L. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Softw. Eng. 22*, 10 (Oct. 1996), 751–761.

[12] Basili, V. R., Shull, F., and Lanubile, F. Building knowledge through families of experiments. *IEEE Trans. Softw. Eng. 25*, 4 (July 1999), 456–473.

[13] Bates, S., and Horwitz, S. Incremental program testing using program dependence graphs. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1993), POPL '93, ACM, pp. 384–396.

[14] Bettenburg, N., Just, S., Schröter, A., Weiss, C., Premraj, R., and Zimmermann, T. What makes a good bug report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering* (New York, NY, USA, 2008), SIGSOFT '08/FSE-16, ACM, pp. 308–318.

[15] Bettenburg, N., Just, S., Schröter, A., Weiss, C., Premraj, R., and Zimmermann, T. Quality of bug reports in eclipse. In *Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology eXchange* (New York, NY, USA, October 2007), ACM Press.

[16] Bettenburg, N., Premraj, R., Zimmermann, T., and Kim, S. Duplicate bug reports considered harmful... really? In *Proceedings of the 24th IEEE International Conference on Software Maintenance* (September 2008).

[17] Bettenburg, N., Premraj, R., Zimmermann, T., and Kim, S. Extracting structural information from bug reports. In *Proceedings of the 2008 international working conference on Mining software repositories* (New York, NY, USA, 2008), MSR '08, ACM, pp. 27–30.

[18] Bhattacharya, P. Using software evolution history to facilitatetate development and maintenance. In *Proceeding of the 33rd international conference on Software engineering* (2011), ACM, pp. 1122–1123.

[19] Bhattacharya, P., and Neamtiu, I. Bug-fix time prediction models: can we do better? In *Proceedings of the 8th Working Conference on Mining Software Repositories* (New York, NY, USA, 2011), MSR '11, ACM, pp. 207–210.

[20] Bird, C., Bachmann, A., Aune, E., Duffy, J., Bernstein, A., Filkov, V., and Devanbu, P. Fair and balanced?: bias in bug-fix datasets. 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* (New York, NY, USA, 2009), ESEC/FSE '09, ACM, pp. 121–130.

[21] Bird, C., Bachmann, A., Rahman, F., and Bernstein, A. Linkster: enabling efficient manual inspection and annotation of mined data. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering* (New York, NY, USA, 2010), FSE '10, ACM, pp. 369–370.

[22] Bird, C., Nagappan, N., Gall, H., Murphy, B., and Devanbu, P. Putting it all together: Using socio-technical networks to predict failures. In *Proceedings of the 2009 20th International Symposium on Software Reliability Engineering* (Washington, DC, USA, 2009), ISSRE '09, IEEE Computer Society, pp. 109–119.

[23] Bird, C., Nagappan, N., Murphy, B., Gall, H., and Devanbu, P. Don't touch my code!: examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering* (New York, NY, USA, 2011), ESEC/FSE '11, ACM, pp. 4–14.

[24] Boetticher, G., T., M., and T., O. Promise repository of empirical software engineering data. online, 2007. Available at: http://promisedata.org/ repository.

[25] Bonacich, P. Power and centrality: A family of measures. *American journal of sociology* (1987).

[26] Briand, L. C., Wüst, J., Ikonomovski, S. V., and Lounis, H. Investigating quality factors in object-oriented designs: an industrial case study. In *Proceedings of the 21st international conference on Software engineering* (New York, NY, USA, 1999), ICSE '99, ACM, pp. 345–354.

[27] Brudaru, I. I., and Zeller, A. What is the long-term impact of changes? In *Proceedings of the 2008 international workshop on Recommendation systems for software engineering* (New York, NY, USA, 2008), RSSE '08, ACM, pp. 30–32.

[28] Burt, R. S. *Structural holes: The social structure of competition.* Harvard University Press, Cambridge, MA, 1992.

[29] Canfora, G., Ceccarelli, M., Cerulo, L., and Di Penta, M. Using multivariate time series and association rules to detect logical change coupling: An empirical study. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance* (Washington, DC, USA, 2010), ICSM '10, IEEE Computer Society, pp. 1–10.

[30] Canfora, G., and Cerulo, L. Fine grained indexing of software repositories to support impact analysis. In *Proceedings of the 2006 international workshop on Mining software repositories* (New York, NY, USA, 2006), MSR '06, ACM, pp. 105–111.

[31] Canfora, G., and Cerulo, L. Supporting change request assignment in open source development. In *Proceedings of the 2006 ACM symposium on Applied computing* (New York, NY, USA, 2006), SAC '06, ACM, pp. 1767–1772.

[32] Chatterjee, K., de Alfaro, L., Raman, V., and Sánchez, C. Analyzing the impact of change in multi-threaded programs. In *Proceedings of the 13th international conference on Fundamental Approaches to Software Engineering* (Berlin, Heidelberg, 2010), FASE'10, Springer-Verlag, pp. 293–307.

[33] Chidamber, S. R., and Kemerer, C. F. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng. 20*, 6 (June 1994), 476–493.

[34] Clarke, E. M., and Emerson, E. A. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop* (London, UK, UK, 1982), Springer-Verlag, pp. 52–71.

[35] Clarke, E. M., Emerson, E. A., and Sistla, A. P. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst. 8*, 2 (Apr. 1986), 244–263.

[36] Csardi, G., and Nepusz, T. The igraph software package for complex network research. *InterJournal Complex Systems* (2006), 1695.

[37] Dagenais, B., and Hendren, L. Enabling static analysis for partial java programs. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications* (New York, NY, USA, 2008), OOPSLA '08, ACM, pp. 313–328.

[38] DALLMEIER, V. *Mining and Checking Object Behavior.* PhD thesis, Universität des Saarlandes, August 2010.

[39] DALLMEIER, V., AND ZIMMERMANN, T. Extraction of bug localization benchmarks from history. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering* (2007), ASE '07, ACM, pp. 433–436.

[40] D'AMBROS, M., LANZA, M., AND ROBBES, R. An extensive comparison of bug prediction approaches. In *MSR* (2010), pp. 31–41.

[41] DENARO, G., AND PEZZÈ, M. An empirical evaluation of fault-proneness models. In *Proceedings of the 24th International Conference on Software Engineering* (New York, NY, USA, 2002), ICSE '02, ACM, pp. 241–251.

[42] DÍAZ, J., PÉREZ, J., GARBAJOSA, J., AND WOLF, A. L. Change impact analysis in product-line architectures. In *Proceedings of the 5th European conference on Software architecture* (Berlin, Heidelberg, 2011), ECSA'11, Springer-Verlag, pp. 114–129.

[43] FENTON, N. E., AND NEIL, M. A critique of software defect prediction models. *IEEE Trans. Softw. Eng. 25*, 5 (Sept. 1999), 675–689.

[44] FERRANTE, J., OTTENSTEIN, K. J., AND WARREN, J. D. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst. 9*, 3 (July 1987), 319–349.

[45] FISCHER, M., PINZGER, M., AND GALL, H. Populating a release history database from version control and bug tracking systems. In *Proceedings of the International Conference on Software Maintenance* (Washington, DC, USA, 2003), ICSM '03, IEEE Computer Society, pp. 23–32.

[46] FLURI, B., AND GALL, H. C. Classifying change types for qualifying change couplings. In *Proceedings of the 14th IEEE International Conference on Program Comprehension* (Washington, DC, USA, 2006), ICPC '06, IEEE Computer Society, pp. 35–45.

[47] FLURI, B., GALL, H. C., AND PINZGER, M. Fine-grained analysis of change couplings. In *Proceedings of the Fifth IEEE International Workshop on Source Code Analysis and Manipulation* (Washington, DC, USA, 2005), SCAM '05, IEEE Computer Society, pp. 66–74.

[48] Fluri, B., Giger, E., and Gall, H. C. Discovering patterns of change types. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering* (Washington, DC, USA, 2008), ASE '08, IEEE Computer Society, pp. 463–466.

[49] Fluri, B., Wuersch, M., PInzger, M., and Gall, H. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Softw. Eng. 33*, 11 (Nov. 2007), 725–743.

[50] Gall, H., Hajek, K., and Jazayeri, M. Detection of logical coupling based on product release history. In *Proceedings of the International Conference on Software Maintenance* (Washington, DC, USA, 1998), ICSM '98, IEEE Computer Society, pp. 190–.

[51] Gall, H., Jazayeri, M., and Krajewski, J. Cvs release history data for detecting logical couplings. In *Proceedings of the 6th International Workshop on Principles of Software Evolution* (Washington, DC, USA, 2003), IWPSE '03, IEEE Computer Society, pp. 13–.

[52] Geiger, R., Fluri, B., Gall, H. C., and Pinzger, M. Relation of code clones and change couplings. In *Proceedings of the 9th international conference on Fundamental Approaches to Software Engineering* (Berlin, Heidelberg, 2006), FASE'06, Springer-Verlag, pp. 411–425.

[53] German, D. Automating the measurement of open source projects. In *In Proceedings of the 3rd Workshop on Open Source Software Engineering* (2003), pp. 63–67.

[54] German, D. M., Hassan, A. E., and Robles, G. Change impact graphs: Determining the impact of prior code changes. *Inf. Softw. Technol. 51*, 10 (Oct. 2009), 1394–1408.

[55] Giger, E., Pinzger, M., and Gall, H. Predicting the fix time of bugs. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering* (New York, NY, USA, 2010), RSSE '10, ACM, pp. 52–56.

[56] Giger, E., Pinzger, M., and Gall, H. C. Comparing fine-grained source code changes and code churn for bug prediction. In *Proceedings of the 8th Working Conference on Mining Software Repositories* (New York, NY, USA, 2011), MSR '11, ACM, pp. 83–92.

[57] Giger, E., Pinzger, M., and Gall, H. C. Can we predict types of code changes? an empirical analysis. In *The 9th Working Conference on Mining Software Repositories* (June 2012), MSR'12, IEEE, pp. 217–226.

[58] Graves, T. L., Karr, A. F., Marron, J. S., and Siy, H. Predicting fault incidence using software change history. *IEEE Trans. Softw. Eng. 26*, 7 (July 2000), 653–661.

[59] Guo, P. J., Zimmermann, T., Nagappan, N., and Murphy, B. Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1* (New York, NY, USA, 2010), ICSE '10, ACM, pp. 495–504.

[60] Guo, P. J., Zimmermann, T., Nagappan, N., and Murphy, B. "not my bug!" and other reasons for software bug report reassignments. In *Proceedings of the ACM 2011 conference on Computer supported cooperative work* (New York, NY, USA, 2011), CSCW '11, ACM, pp. 395–404.

[61] Gurfinkel, A., Devereux, B., and Chechik, M. Model exploration with temporal logic query checking. *SIGSOFT Softw. Eng. Notes 27*, 6 (Nov. 2002), 139–148.

[62] Harrold, M. J., Malloy, B., and Rothermel, G. Efficient construction of program dependence graphs. *SIGSOFT Softw. Eng. Notes 18*, 3 (July 1993), 160–170.

[63] Hassan, A. E. Automated classification of change messages in open source projects. In *Proceedings of the 2008 ACM symposium on Applied computing* (New York, NY, USA, 2008), SAC '08, ACM, pp. 837–841.

[64] Hassan, A. E. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering* (Washington, DC, USA, 2009), ICSE '09, IEEE Computer Society, pp. 78–88.

[65] Hassan, A. E., and Holt, R. C. The top ten list: Dynamic fault prediction. In *Proceedings of the 21st IEEE International Conference on Software Maintenance* (Washington, DC, USA, 2005), ICSM '05, IEEE Computer Society, pp. 263–272.

[66] Hassan, A. E., and Holt, R. C. Replaying development history to assess the effectiveness of change propagation tools. *Empirical Softw. Engg. 11*, 3 (Sept. 2006), 335–367.

[67] HERZIG, K., JUST, S., RAU, A., AND ZELLER, A. Classifying Changes and Predicting Defects using Change Genealogies. Technical report, Saarland University, 2012.

[68] HERZIG, K., JUST, S., AND ZELLER, A. It's not a Bug, it's a Feature: How Misclassification Impacts Bug Prediction. In *Proceedings of the 35nd ACM/IEEE International Conference on Software Engineering - Volume 1* (New York, NY, USA, 2013), ICSE '13, ACM.

[69] HERZIG, K., AND ZELLER, A. Mining the Jazz repository: Challenges and Opportunities. In *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories* (Washington, DC, USA, 2009), MSR '09, IEEE Computer Society, pp. 159–162.

[70] HERZIG, K., AND ZELLER, A. *Mining Your Own Evidence*. O'Reilly Media, October 2010, ch. Mining Your Own Evidence, pp. 517–529.

[71] HERZIG, K., AND ZELLER, A. Mining Cause-Effect-Chains from Version Histories. In *Proceedings of the 2011 IEEE 22nd International Symposium on Software Reliability Engineering* (Washington, DC, USA, 2011), ISSRE '11, IEEE Computer Society, pp. 60–69.

[72] HERZIG, K., AND ZELLER, A. Untangling Changes. Technical report, Saarland University, 2011.

[73] HERZIG, K. S. Capturing the long-term impact of changes. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2* (New York, NY, USA, 2010), ICSE '10, ACM, pp. 393–396.

[74] HOLSCHUH, T., PAEUSER, M., HERZIG, K., ZIMMERMANN, T., PREMRAJ, R., AND ZELLER, A. Predicting defects in sap java code: An experience report. In *Proceedings of the 31th International Conference on Software Engineering* (May 2009).

[75] HOOIMEIJER, P., AND WEIMER, W. Modeling bug report quality. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering* (New York, NY, USA, 2007), ASE '07, ACM, pp. 34–43.

[76] JACCARD, P. Étude comparative de la distribution florale dans une portion des Alpes et des Jura. *Bulletin del la Société Vaudoise des Sciences Naturelles 37* (1901), 547–579.

[77] JASHKI, M.-A., ZAFARANI, R., AND BAGHERI, E. Towards a more efficient static software change impact analysis method. In *Proceedings of the 8th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering* (New York, NY, USA, 2008), PASTE '08, ACM, pp. 84–90.

[78] Ibm Rational Jazz. Available at: http://jazz.net/.

[79] KAFURA, D., AND REDDY, G. R. The use of software complexity metrics in software maintenance. *IEEE Trans. Softw. Eng. 13*, 3 (Mar. 1987), 335–343.

[80] KARYPIS, G., AND KUMAR, V. Analysis of multilevel graph partitioning. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing* (1995), Supercomputing 1995, ACM.

[81] KARYPIS, G., AND KUMAR, V. *MeTis: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0*, 1995.

[82] KARYPIS, G., AND KUMAR, V. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput. 20* (December 1998), 359–392.

[83] KAWRYKOW, D. Enabling precise interpretations of software change data. Master's thesis, McGill University, August 2011.

[84] KAWRYKOW, D., AND ROBILLARD, M. P. Non-essential changes in version histories. In *Proceedings of the 33rd International Conference on Software Engineering* (New York, NY, USA, 2011), ICSE '11, ACM, pp. 351–360.

[85] KIM, M., AND NOTKIN, D. Discovering and representing systematic code changes. In *Proceedings of the 31st International Conference on Software Engineering* (Washington, DC, USA, 2009), ICSE '09, IEEE Computer Society, pp. 309–319.

[86] KIM, S., E. JAMES WHITEHEAD, J., AND ZHANG, Y. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering 34*, 2 (March/April 2008), 181–196.

[87] KIM, S., ZHANG, H., WU, R., AND GONG, L. Dealing with noise in defect prediction. In *Proceedings of the 33rd International Conference on Software Engineering* (New York, NY, USA, 2011), ICSE '11, ACM, pp. 481–490.

[88] KIM, S., ZIMMERMANN, T., KIM, M., HASSAN, A., MOCKUS, A., GIRBA, T., PINZGER, M., WHITEHEAD, JR., E. J., AND ZELLER, A. Ta-re: an exchange language for mining software repositories. In *Proceedings of the 2006 international workshop*

on Mining software repositories (New York, NY, USA, 2006), MSR '06, ACM, pp. 22–25.

[89] Kim, S., Zimmermann, T., Whitehead Jr., E. J., and Zeller, A. Predicting faults from cached history. In *Proceedings of the 29th international conference on Software Engineering* (Washington, DC, USA, 2007), ICSE '07, IEEE Computer Society, pp. 489–498.

[90] Kuhn, M. *caret: Classification and Regression Training*, 2011. R package version 4.76.

[91] Law, J., and Rothermel, G. Whole program path-based dynamic impact analysis. In *Proceedings of the 25th International Conference on Software Engineering* (Washington, DC, USA, 2003), ICSE '03, IEEE Computer Society, pp. 308–318.

[92] Lee, M., Offutt, A. J., and Alexander, R. T. Algorithmic analysis of the impacts of changes to object-oriented software. In *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 34'00)* (Washington, DC, USA, 2000), TOOLS '00, IEEE Computer Society, pp. 61–.

[93] Li, P. L., Kivett, R., Zhan, Z., Jeon, S.-e., Nagappan, N., Murphy, B., and Ko, A. J. Characterizing the differences between pre- and post-release versions of software. In *Proceeding of the 33rd international conference on Software engineering* (2011), ACM, pp. 716–725.

[94] Liebchen, G. A., and Shepperd, M. Data sets and data quality in software engineering. In *Proceedings of the 4th international workshop on Predictor models in software engineering* (New York, NY, USA, 2008), PROMISE '08, ACM, pp. 39–44.

[95] Marks, L., Zou, Y., and Hassan, A. E. Studying the fix-time for bugs in large open source projects. In *Proceedings of the 7th International Conference on Predictive Models in Software Engineering* (New York, NY, USA, 2011), Promise '11, ACM, pp. 11:1–11:8.

[96] McCabe, T. J. A complexity measure. *IEEE Trans. Software Eng. 2*, 4 (1976), 308–320.

[97] McConnell, S. *Code complete: a practical handbook of software construction*. Microsoft Press, Redmond, WA, USA, 1993.

[98] Menzies, T., Milton, Z., Turhan, B., Cukic, B., Jiang, Y., and Bener, A. Defect prediction from static code features: current results, limitations, new approaches. *Automated Software Engg. 17*, 4 (Dec. 2010), 375–407.

[99] Mockus, A. Missing data in software engineering. *Guide to Advanced Empirical Software Engineering* (2008), 185–200.

[100] Mockus, A., Nagappan, N., and Dinh-Trong, T. T. Test coverage and post-verification defects: A multiple case study. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement* (Washington, DC, USA, 2009), ESEM '09, IEEE Computer Society, pp. 291–301.

[101] Mockus, A., and Votta, L. G. Identifying reasons for software changes using historic databases. In *Proceedings of the International Conference on Software Maintenance (ICSM'00)* (Washington, DC, USA, 2000), ICSM '00, IEEE Computer Society, pp. 120–.

[102] Moser, R., Pedrycz, W., and Succi, G. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th international conference on Software engineering* (New York, NY, USA, 2008), ICSE '08, ACM, pp. 181–190.

[103] Murgia, A., Concas, G., Marchesi, M., and Tonelli, R. A machine learning approach for text categorization of fixing-issue commits on cvs. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement* (New York, NY, USA, 2010), ESEM '10, ACM, pp. 6:1–6:10.

[104] Nagappan, N., and Ball, T. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th international conference on Software engineering* (New York, NY, USA, 2005), ICSE '05, ACM, pp. 284–292.

[105] Nagappan, N., and Ball, T. Using software dependencies and churn metrics to predict field failures: An empirical case study. In *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement* (Washington, DC, USA, 2007), ESEM '07, IEEE Computer Society, pp. 364–373.

[106] Nagappan, N., and Ball, T. *Evidence-Based Failure Prediction*. O'Reilly Media, October 2010, ch. Evidence-Based Failure Prediction, pp. 415–434.

[107] Nagappan, N., Ball, T., and Murphy, B. Using historical in-process and product metrics for early estimation of software failures. In *Proceedings of the 17th International Symposium on Software Reliability Engineering* (Washington, DC, USA, 2006), ISSRE '06, IEEE Computer Society, pp. 62–74.

[108] Nagappan, N., Ball, T., and Zeller, A. Mining metrics to predict component failures. In *Proceedings of the 28th international conference on Software engineering* (New York, NY, USA, 2006), ICSE '06, ACM, pp. 452–461.

[109] Nagappan, N., Murphy, B., and Basili, V. The influence of organizational structure on software quality: an empirical case study. In *Proceedings of the 30th international conference on Software engineering* (New York, NY, USA, 2008), ICSE '08, ACM, pp. 521–530.

[110] Nagappan, N., Zeller, A., Zimmermann, T., Herzig, K., and Murphy, B. Change bursts as defect predictors. In *Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering* (Washington, DC, USA, 2010), ISSRE '10, IEEE Computer Society, pp. 309–318.

[111] Nguyen, T. H. D., Adams, B., and Hassan, A. E. A case study of bias in bug-fix datasets. In *Proceedings of the 2010 17th Working Conference on Reverse Engineering* (Washington, DC, USA, 2010), WCRE '10, IEEE Computer Society, pp. 259–268.

[112] Ohlsson, N., and Alberg, H. Predicting fault-prone software modules in telephone switches. *IEEE Trans. Softw. Eng. 22*, 12 (Dec. 1996), 886–894.

[113] Orso, A., Apiwattanapong, T., and Harrold, M. J. Leveraging field data for impact analysis and regression testing. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering* (New York, NY, USA, 2003), ESEC/FSE-11, ACM, pp. 128–137.

[114] Ostrand, T. J., Weyuker, E. J., and Bell, R. M. Where the bugs are. In *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis* (New York, NY, USA, 2004), ISSTA '04, ACM, pp. 86–96.

[115] Ostrand, T. J., Weyuker, E. J., and Bell, R. M. Predicting the location and number of faults in large software systems. *IEEE Trans. Softw. Eng. 31*, 4 (Apr. 2005), 340–355.

[116] PINZGER, M., NAGAPPAN, N., AND MURPHY, B. Can developer-module networks predict failures? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering* (New York, NY, USA, 2008), SIGSOFT '08/FSE-16, ACM, pp. 2–12.

[117] PODGURSKI, A., AND CLARKE, L. A. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Trans. Softw. Eng. 16*, 9 (Sept. 1990), 965–979.

[118] POLO, M., PIATTINI, M., AND RUIZ, F. Using code metrics to predict maintenance of legacy programs: A case study. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)* (Washington, DC, USA, 2001), ICSM '01, IEEE Computer Society, pp. 202–.

[119] PREMRAJ, R., AND HERZIG, K. Network versus code metrics to predict defects: A replication study. In *Proceedings of the 2011 International Symposium on Empirical Software Engineering and Measurement* (Washington, DC, USA, 2011), ESEM '11, IEEE Computer Society, pp. 215–224.

[120] R DEVELOPMENT CORE TEAM. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, 2010.

[121] REN, X., SHAH, F., TIP, F., RYDER, B. G., AND CHESLEY, O. Chianti: a tool for change impact analysis of java programs. *SIGPLAN Not. 39*, 10 (Oct. 2004), 432–448.

[122] RIAZ, M., MENDES, E., AND TEMPERO, E. A systematic review of software maintainability prediction and metrics. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement* (Washington, DC, USA, 2009), ESEM '09, IEEE Computer Society, pp. 367–377.

[123] ROTHERMEL, G., AND HARROLD, M. J. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Methodol. 6*, 2 (Apr. 1997), 173–210.

[124] RUNESON, P., ALEXANDERSSON, M., AND NYHOLM, O. Detection of duplicate defect reports using natural language processing. In *Proceedings of the 29th international conference on Software Engineering* (Washington, DC, USA, 2007), ICSE '07, IEEE Computer Society, pp. 499–510.

[125] SALMAN, N., AND DOĞRU, A. Design effort estimation using complexity metrics. *J. Integr. Des. Process Sci. 8*, 3 (Aug. 2004), 83–88.

[126] Schröter, A., Zimmermann, T., and Zeller, A. Predicting component failures at design time. In *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering* (New York, NY, USA, 2006), ISESE '06, ACM, pp. 18–27.

[127] Sherriff, M., and Williams, L. Empirical software change impact analysis using singular value decomposition. In *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation* (Washington, DC, USA, 2008), ICST '08, IEEE Computer Society, pp. 268–277.

[128] Shin, Y., Bell, R., Ostrand, T., and Weyuker, E. Does calling structure information improve the accuracy of fault prediction? In *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories* (Washington, DC, USA, 2009), MSR '09, IEEE Computer Society, pp. 61–70.

[129] Śliwerski, J., Zimmermann, T., and Zeller, A. When do changes induce fixes? In *Proceedings of the 2005 international workshop on Mining software repositories* (2005), MSR '05, ACM, pp. 1–5.

[130] Somasundaram, K., and Murphy, G. C. Automatic categorization of bug reports using latent dirichlet allocation. In *Proceedings of the 5th India Software Engineering Conference* (New York, NY, USA, 2012), ISEC '12, ACM, pp. 125–130.

[131] Song, Q., Shepperd, M., and Mair, C. Using grey relational analysis to predict software effort with small data sets. In *Proceedings of the 11th IEEE International Software Metrics Symposium* (Washington, DC, USA, 2005), METRICS '05, IEEE Computer Society, pp. 35–.

[132] Stoerzer, M., Ryder, B. G., Ren, X., and Tip, F. Finding failure-inducing changes in java programs using change classification. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering* (New York, NY, USA, 2006), SIGSOFT '06/FSE-14, ACM, pp. 57–68.

[133] Tang, A., Nicholson, A., Jin, Y., and Han, J. Using bayesian belief networks for change impact analysis in architecture design. *J. Syst. Softw. 80* (January 2007), 127–148.

[134] Tosun, A., Turhan, B., and Bener, A. Validation of network measures as indicators of defective modules in software systems. In *Proceedings of the 5th International Conference on Predictor Models in Software Engineering* (New York, NY, USA, 2009), PROMISE '09, ACM, pp. 5:1–5:9.

[135] Čubranić, D. Automatic bug triage using text categorization. In *In SEKE 2004: Proceedings of the Sixteenth International Conference on Software Engineering and Knowledge Engineering* (2004), KSI Press, pp. 92–97.

[136] Čubranić, D., and Murphy, G. C. Hipikat: recommending pertinent software development artifacts. In *Proceedings of the 25th International Conference on Software Engineering* (Washington, DC, USA, 2003), ICSE '03, IEEE Computer Society, pp. 408–418.

[137] Čubranić, D., Murphy, G. C., Singer, J., and Booth, K. S. Hipikat: A project memory for software development. *IEEE Trans. Softw. Eng. 31*, 6 (June 2005), 446–465.

[138] Wang, X., Zhang, L., Xie, T., Anvik, J., and Sun, J. An approach to detecting duplicate bug reports using natural language and execution information. In *Proceedings of the 30th international conference on Software engineering* (New York, NY, USA, 2008), ICSE '08, ACM, pp. 461–470.

[139] Wasylkowski, A. *Object Usage: Patterns and Anomalies*. PhD thesis, Saarland University, Sept. 2010.

[140] Wasylkowski, A., and Zeller, A. Mining temporal specifications from object usage. *Automated Software Engg. 18*, 3-4 (Dec. 2011), 263–292.

[141] Weiss, C., Premraj, R., Zimmermann, T., and Zeller, A. How long will it take to fix this bug? In *Proceedings of the Fourth International Workshop on Mining Software Repositories* (Washington, DC, USA, 2007), MSR '07, IEEE Computer Society, pp. 1–.

[142] Williams, B. J., and Carver, J. C. Characterizing software architecture changes: A systematic review. *Inf. Softw. Technol. 52* (January 2010), 31–51.

[143] Witten, I. H., and Frank, E. Data mining: practical machine learning tools and techniques with java implementations. *SIGMOD Rec. 31*, 1 (Mar. 2002), 76–77.

[144] Wloka, J., Ryder, B., Tip, F., and Ren, X. Safe-commit analysis to facilitate team software development. In *Proceedings of the 31st International Conference on Software Engineering* (Washington, DC, USA, 2009), ICSE '09, IEEE Computer Society, pp. 507–517.

[145] Wu, R., Zhang, H., Kim, S., and Cheung, S.-C. 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 (New York, NY, USA, 2011), ESEC/FSE '11, ACM, pp. 15–25.

[146] YING, A. T. T. Predicting source code changes by mining revision history, 2003.

[147] ZENG, H., AND RINE, D. Estimation of software defects fix effort using neural networks. In *Proceedings of the 28th Annual International Computer Software and Applications Conference - Workshops and Fast Abstracts - Volume 02* (Washington, DC, USA, 2004), COMPSAC '04, IEEE Computer Society, pp. 20–21.

[148] ZIMMERMAN, T., NAGAPPAN, N., HERZIG, K., PREMRAJ, R., AND WILLIAMS, L. An empirical study on the relation between dependency neighborhoods and failures. In *Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation* (Washington, DC, USA, 2011), ICST '11, IEEE Computer Society, pp. 347–356.

[149] ZIMMERMANN, T., DIEHL, S., AND ZELLER, A. How history justifies system architecture (or not). In *Proceedings of the 6th International Workshop on Principles of Software Evolution* (Washington, DC, USA, 2003), IWPSE '03, IEEE Computer Society, pp. 73–.

[150] ZIMMERMANN, T., AND NAGAPPAN, N. Predicting subsystem failures using dependency graph complexities. In *Proceedings of the The 18th IEEE International Symposium on Software Reliability* (Washington, DC, USA, 2007), ISSRE '07, IEEE Computer Society, pp. 227–236.

[151] ZIMMERMANN, T., AND NAGAPPAN, N. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th international conference on Software engineering* (New York, NY, USA, 2008), ICSE '08, ACM, pp. 531–540.

[152] ZIMMERMANN, T., NAGAPPAN, N., GALL, H., GIGER, E., AND MURPHY, B. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. 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* (New York, NY, USA, 2009), ESEC/FSE '09, ACM, pp. 91–100.

[153] ZIMMERMANN, T., PREMRAJ, R., AND ZELLER, A. Predicting defects for Eclipse. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering* (Washington, DC, USA, 2007), PROMISE '07, IEEE Computer Society, pp. 9–.

[154] ZIMMERMANN, T., WEISGERBER, P., DIEHL, S., AND ZELLER, A. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering* (Washington, DC, USA, 2004), ICSE '04, IEEE Computer Society, pp. 563–572.

Glossary

Bug A software *bug* refers to a fault in the source code of a program that caused a software failure. Fixing a bug means to improve the source code in a way that a particular software failure, often detected by observing a software error, is overcome.

Bug report The term *bug report* is a bit unfortunate and misleading. A bug report is an issue report that files an observed error or a detected failure. Bug reports filed by non-developers (e.g. customers) are based on the observation of an error. It is not uncommon that bug reports are not indicating a bug in the source code but are due to wrong expectations, simple misconfiguration, or likewise. Therefore the term error report would have been more accurate.

Change dependency rule A set of rules used to compute dependencies between individual *change operations*. For more details see Section 3.2.

Change genealogy A graph structure to model dependencies between change sets.

Change genealogy layer A virtual *change genealogy* lifting the granularity level of the original change genealogy to a level above the change operation level. For more details see Section 3.4.

Change genealogy metric A *network metric* based on *change genealogy* graph structures modeling dependencies between individual code changes instead of dependencies between code artifacts.

Change operation A set of source code changes that add, modify, or delete method calls or method definitions. For more details see Section 3.5.1.

Change set A set of code changes simultaneously committed to the version archive. Sometime also called commits or transactions.

Change set partition A subset of *change operations* in a *change set* which serve one individual *developer maintenance task*.

Complexity metric Classical set of *software metrics* that indicate the complexity of a code artifact using the artifacts code structure.

Defect In this thesis, we use the term software *defect* as a synonym for software bug. In the testing community, the term software defect often refers to mismatches between requirements.

Developer maintenance task A modification of a software product performed after delivery to correct, adapt, enhance, or prevent a software system due to one independent issue.

Error The discrepancy between a computed and an observed value.

Failure The inability of a program to perform its requires, expected, or specified task.

Functional quality Functional quality reflects the fraction of met required and specified functionality.

Change history metric Set of software metrics that describe the change history of the code (e.g. the number of applied changes in the past).

Issue In software development, the term *issue* describes a unit of work to accomplish an improvement of the software system. This can be a problem with the software that needs to be fixed, but it might also be a request to improve the speed or reliability of the software system, or it might be a request to implement a new feature.

Issue report An *issue report* is a record in a software repository that records an issue to be addressed by one or multiple team members. Analogue to the definition of an issue, an issue report can be a bug report, an improvement request, or a feature request.

Issue tracking system Glsplbugtracker are used to track developer tasks and program issues. Prominent examples are BUGZILLA and JIRA.

Network metric Set of *software metrics* derived from network graphs modeling dependencies between source code artifacts.

Quality Refers to *structural quality* unless stated otherwise.

Software repository An umbrella term for repositories and databases containing different kind of artifacts produced during development but also usage of a software product.

Software failure A software *failure* refers to the inability of the program to perform its required, expected, or specified task. A failure can refer to correctness, performance, etc.

Software error An *error* in software refers to a discrepancy between a computed, observed, or measured value and the expected or specified value that should have been computed. Thus, an error indicated that the software is not working as expected or specified. An error cannot be fixed. It can be observed, only.

Software metric Objective, reproducible, and quantifiable measurements of source code.

Software quality Software *quality* refers to two different notions: *functional quality* and *structural quality*. Functional quality refers to functional requirements and specifications. A high functional quality reflects the fact that the software system meets a high fraction of required and specified functionality. In contrast, structural quality refers to non-functional requirements and specifications such as maintainability, stability, robustness, or the number of failures. Thus, the more bugs were found in the source code, the lower the structural quality of the software system—simply because a bug causes a software failure. In this thesis, the term *quality* stands as a synonym for structural quality, unless stated otherwise.

Structural quality Structural quality refers to non-functional requirements and specifications such as maintainability, stability, robustness, or the number of failures.

Tangled change set A *change set* that contains code changes serving multiple *developer maintenance tasks* (e.g. bug fix and feature implementation).

Artificially tangled change set A *tangled change set* that was artificially created to evaluate the untangling algorithm (see Section 5.3).

Version archive / Version control system Version archives or version control systems record changes to applied documents and files. Prominent examples are `Git` or `Subversion`.

Appendix

The Mozkito Framework

For the experiments described in this thesis, we required a general purpose mining framework to mine and analyze version control systems and bug repositories. The construction of change genealogies, change genealogy layers, and the computation of change genealogy metrics requires a flexible add-on infrastructure that allows static source code analysis. Therefore, we developed MOZKITO, a mining framework that can handle projects and repositories of large size. The goal of MOZKITO was to construct and maintain an publicly available open-source mining tool that supports the most popular software repositories and that can be extended easily without modifying the core components. For this reason, MOZKITO is structured in modules. Each experiment described in this thesis is a module that makes used of and can be reused by other modules. To this end, we implemented to following modules and functionalities:

MOZKITO-PERSISTENCE: Many data mining approaches are expensive in terms of runtime. To make results of modules effectively reusable, MOZKITO can persist JAVA objects into a relational database. This persistence layer is an abstraction layer that can be used with most open-source and commercial JAVA persistence libraries such as `OpenJPA`[1] or `Hibernate`[2]. Each module can use JAVA annotations to define its own JAVA objects to be stored in a database. Database tables will be created at the time the corresponding MOZKITO module get loaded.

By default, MOZKITO uses the `OpenJPA` as underlying persistence layer to store JAVA objects into a `PostgreSQL`[3].

[1]`http://openjpa.apache.org`
[2]`http://www.hibernate.org`
[3]`http://www.postgresql.org`

Mozkito-Versions: This module contains functionality to mine, parse, and to persist version control system repositories. For each change set applied to the version control system Mozkito creates and persists Java objects representing the individual components of a change set; such as the id of the change set, the author that applied the change set, the timestamp and commit message and basic information about the source files changes within the change set. Information about individual change operations will be generated and persisted in the Mozkito-PatchAnalysis module.

Mozkito supports the following version control system systems: `git` [1], `mercurial` [3], and `subversion` [4]. To extend the set of supported version control systems it suffices to implement the `Repository` interface and to add a corresponding version control system type to the `RepositoryType` enum.

Mozkito-PatchAnalysis: supplies functionality to extract and persists change operation from change set as described in Section 3.2. The module internally uses the EclipseJava development tools (`JDT`)[4] and the `Partial Program Analysis` tool [37] to map code changes applied by a change set to differences within the Java abstract syntax tree of each changed source file.

Mozkito-Genealogies: contains the complete implementation of change genealogies as described in Chapter 3 using the Mozkito-PatchAnalysis for change operation extraction, including change genealogy layers. Change genealogies are graph structures. Unlike other data structures, change genealogies are not persistent in a relational database. Mozkito stores the basic graph structure in a `Neo4J`[5] graph database. Each graph database node refers to a change set Java object stored within the relational database. For compatibility, visualization, and export reasons, it is possible to transform a change genealogy into a `JUNG`[6] graph data structure.

To create new change genealogy layers it is necessary to add a new Mozkito module or to modify the existing module and to extend the abstract `ChangeGenealogyLayer` class. Change genealogy layers are read only data structures. The two main change genealogy layers (the change operation and the change set layer) are stored in separate `Neo4J` database within the same directory. Change genealogy layers translate database requests one the fly to requests targeting the core change operation change genealogy layer. This on the fly

[4]`http://www.eclipse.org/jdt`
[5]`http://neo4j.org`
[6]`http://jung.sourceforge.net`

translation is slow and increases runtime significantly. To accelerate our experiments on the change set change genealogy layer (our default change genealogy layer) we decided to create a separate `Neo4J` database for both: the core change operation change genealogy layers as well as for the default change set change genealogy layer.

Mozkito-ModelChecking: adds model checking support to change genealogies. This module contains all necessary implementations and tools to reproduce the experiments described in Chapter 7. The module contains a complete model checking engine written by Andrzej Wasylkowski [139] allowing to model check CTL formulas on change genealogies.

Mozkito-GenealogyMetrics: contains implementations of change genealogy metrics as described and discussed in Chapter 6. The module also contains the mechanisms to compute change genealogy metrics for change genealogy vertices. The metric values are written to CSV files. Persisting metric results would make no sense since change genealogy metric values are likely to change as soon as the software project and the corresponding change genealogy graph structure evolves.

Mozkito-Issues: contains a complete tool chain to parse and persist issue report and their comments and report history. Each report is represented by a set of Java objects that contain a snapshot of issue report as it was at the time the report got parsed. Mozkito also persists the changes applied to a issue report so far and allows the user to retrieve a virtual issue report representing the issue report at an earlier point in time.

By default, Mozkito is able to automatically parse the following bug tracking systems: `Bugzilla`[7], `GoogleCode`[8], `Jira`[9], and `Mantis`[10]. Mozkito also contains an implementation of `Sourceforge` bug tracker but the current implementation is outdated and with `Sourceforge` changing its interface very frequently without providing a complete and functional REST API, we dropped the support for `Sourceforge`.

Mozkito-Persons: Version control systems and bug repositories (or any other software repository) contain related artifacts but rarely share the same authentication systems and user databases. When mining and persisting change sets and issue

[7] www.bugzilla.org
[8] http://code.google.com
[9] http://www.atlassian.com/software/jira
[10] www.mantisbt.org

reports Mozkito creates Java objects for every distinct author of an artifact. The model containing the persisted Java objects are contained in this module. Further, when mapping artifacts stemming from different software repositories it is often necessary to check if artifacts are created or modified by the same developer or person. But frequently, one developer has different user names identifying him in different software archives (e.g. a short user name in `git` and the email address in the bug database). The Mozkito-Persons module contains tools and algorithms that allow to merge multiple `Person` instances referring to the same developer (e.g. sharing the same email address, full name, or user name).

Mozkito-Mappings: Contains a complete tool chain that allows to relate change sets with issue report (see Section 2.3.4). The module allows to use multiple simple and complex mapping strategies. Please refer to the Mozkito website for more details.

Mozkito-Callgraph: Contains a tool chain that allows to create and use static call graphs for individual version control system change sets.

Mozkito-ChangeCouplings: Implements change couplings as proposed by Zimmermann et al. [154]. Mozkito-ChangeCouplings allows the computation of change couplings on source file and on Java method level. Please refer to the Mozkito website for more details.

Mozkito-Untangle: implements the *untangling algorithm* described, discussed, and evaluated in Chapter 5.

Mozkito is a public available open source framework published under the *Apache License, Version 2.0.* To download, use, extend, or learn more about Mozkito please visit:

`http://mozkito.org`

Starting Mozkito Processes

By default, Mozkito modules are meant to be stand-alone. The build process of Mozkito will generate an executable stand-alone `jar` file containing all required libraries. Thus, starting a Mozkito process means to run the corresponding `jar` file.

Mozkito comes with a command line framework that allows easy declaration of command line `JAVA-VM` arguments. To list the mandatory and optional command line arguments, add the `-Dhelp` `JAVA-VM` command line argument when starting the process. To show the list of command line arguments for Mozkito-Versions please execute:

```
java -Dhelp -jar moskito-versions-jar-with-dependencies.jar
```

This should print the command line arguments for the specified Mozkito-Versions `jar` file.

Extending Mozkito

The default way to extend Mozkito is to add a new `maven` module to the existing Mozkito `maven` project. For more details on `maven` please refer to

`http://www.sonatype.com/books/mvnex-book/reference/multimodule.html`