14th International Workshop on Principles of Software Evolution (IWPSE 2015)

Proceedings

Ángela Lozano and Gregorio Robles

August 30, 2015
Bergamo, Italy

The Association for Computing Machinery, Inc.
2 Penn Plaza, Suite 701
New York, NY 10121-0701

Copyright © 2015 by the Association for Computing Machinery, Inc (ACM). Permission to make digital or hard copies of portions of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted.

To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permission to republish from: Publications Dept. ACM, Inc.
Fax +1-212-869-0481 or E-mail permissions@acm.org.

For other copying of articles that carry a code at the bottom of the first or last page, copying is permitted provided that the per-copy fee indicated in the code is paid through the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923.

Notice to Past Authors of ACM-Published Articles

ACM intends to create a complete electronic archive of all articles and/or other material previously published by ACM. If you have written a work that was previously published by ACM in any journal or conference proceedings prior to 1978, or any SIG Newsletter at any time, and you do NOT want this work to appear in the ACM Digital Library, please inform permissions@acm.org, stating the title of the work, the author(s), and where and when published.

ACM ISBN: 978-1-4503-3816-5

Additional copies may be ordered prepaid from:

	Phone: 1-800-342-6626
ACM Order Department	(U.S.A. and Canada)
P.O. BOX 11405	+1-212-626-0500
Church Street Station	(All other countries)
New York, NY 10286-1405	Fax: +1-212-944-1318
	E-mail: acmhelp@acm.org

Production: Conference Publishing Consulting
D-94034 Passau, Germany, info@conference-publishing.com

Message from the Chairs

Welcome to the 14th International Workshop on Principles of Software Evolution (IWPSE'15) co-located with ESEC/FSE 15, August 30th 2015 in Bergamo, Italy. IWPSE'15 groups high-quality papers presenting experiments, surveys, approaches, techniques and tools related to the evolution of software systems.

Research in software evolution and evolvability has been thriving in the past years, with a constant stream of new formalisms, tools, techniques, and development methodologies. Research in software evolution has two goals. The first is to facilitate the way software systems can be changed so they become long-lived; this includes coping with demands from users and with the increasing complexity and volatility of contexts in which such systems may operate. The second goal is to understand and if possible control the processes by which demand for these changes come about.

Topics of interest include, but are not limited to:

- Application areas: distributed, embedded, real-time, ultra large scale, and self-adaptive systems, web services, mobile computing, information systems, systems of systems, etc.
- Paradigms: support and barriers to evolution in aspect-oriented, agile, component-based, and model-driven software development, service-oriented architectures,etc.
- Technical aspects: co-evolution and inconsistency management, impact analysis and change propagation, dynamic reconfiguration and updating; architectures, tools, languages and notations for supporting evolution, etc.
- Managerial aspects: effort and cost estimation, risk analysis, software quality, productivity, process support, training, awareness, etc.
- Empirical studies related to software evolution.
- Mining software repositories approaches and techniques supporting software evolution.
- Industrial experience on successes and failures related to software evolution.
- Interdisciplinary approaches: adaptation of evolutionary concepts and measures from other disciplines (biology, geology, etc.) to software evolution.
- Theories and models to explain and understand software evolution.

Given the increasing importance of empirical studies in our field, we found it relevant to devote the first session to hear about the pitfalls and achievements of empirical studies in software engineering from one of the most experienced researchers on Empirical Software Engineering, Prof. Sandro Morasca (from Università degli Studi dell'Insubria, Italy).

This year, we received a total of 13 submissions (11 research papers and 2 position papers) from Algeria, Belgium, Canada, Finland, France, Italy, Japan, Malta, Netherlands, Pakistan, Sweden, and United Kingdom. From this we selected 9 research papers. Each paper was thoroughly reviewed by at least three reviewers, followed by an open discussion in EasyChair where program committee members had the opportunity to discuss all papers except those for which they had a conflict of interest. After the

keynote talk, we will have three sessions: Refactoring and Testing, APIs and Human Factors, and Analysis Techniques. Each session presenting three papers.

We would like to thank the authors for the high quality papers they submitted, and the program committee members and additional reviewers for their detailed reviews submitted on time in our tight schedule. We hope that you will find the program interesting and we look forward to meeting everyone at the workshop in Bergamo.

Gregorio Robles and Ángela Lozano
IWPSE'15 Programme Co-chairs

IWPSE'15 Organization

Organizing Committee

Program Co-Chairs

Ángela Lozano	Vrije Universiteit Brussel, Belgium
Gregorio Robles	Universidad Rey Juan Carlos, Spain

Steering Committee

Michel Wermelinger (chair)	The Open University, UK
Gerardo Canfora	University of Sannio, Italy
Andrea Capiluppi	University of East London, UK
Anthony Cleve	University of Namur, Belgium
Massimiliano di Penta	University of Sannio, Italy
Michele Lanza	University of Lugano, Switzerland
Ángela Lozano	Vrije Universiteit Brussel, Belgium
Kim Mens	Université Catholique de Louvain, Belgium
Tom Mens	University of Mons, Belgium
Naouel Moha	University of Québec in Montréal, Canada
Romain Robbes	University of Chile, Chile
Gregorio Robles	Universidad Rey Juan Carlos, Spain
Motoshi Saeki	Tokyo Institute of Technology, Japan

Program Committee

Bram Adams	Queen's University, Canada
Tom Arbuckle	University of Limerick, Ireland
Árpád Beszédes	University of Szeged, Hungary
Rafael Capilla	Universidad Rey Juan Carlos, Spain
Andrea Capiluppi	University of East London, UK
Massimiliano Di Penta	University of Sannio, Italy
Tamás Gergely	University of Szeged, Hungary
Emanuel Giger	University of Zurich, Switzerland
Mike Godfrey	University of Waterloo, Canada
Andrian Marcus	Wayne State University, USA
Alessandro Murgia	University of Antwerp, Belgium
Carlos Noguera	Vrije Universiteit Brussel, Belgium
Martin Pinzger	Alpen-Adria Universität, Austria

Denys Poshyvanyk	The College of William and Mary, USA
Daniel Rodriguez	University of Alcalá, Spain
Mark Van Den Brand	Eindhoven University of Technology, The Netherlands
Michel Wermelinger	The Open University, UK
Andy Zaidman	Delft University of Technology, The Nertherlands

Additional Reviewers

Venera Arnaoudova
Yanja Dajsuren
Laura Moreno
David Tengeri

Contents

Frontmatter

Foreword . . . iii

Refactoring and Testing

Localising Faults in Test Execution Traces
Gulsher Laghari, Alessandro Murgia, and Serge Demeyer — *University of Antwerp, Belgium* . . . 1

Circumventing Refactoring Masking using Fine-Grained Change Recording
Quinten David Soetens, Javier Pérez, Serge Demeyer, and Andy Zaidman — *University of Antwerp, Belgium; Delft University of Technology, Netherlands* . . . 9

Hierarchical Categorization of Edit Operations for Separately Committing Large Refactoring Results
Jumpei Matsuda, Shinpei Hayashi, and Motoshi Saeki — *Tokyo Institute of Technology, Japan* . . . 19

APIs and Human Factors

The Driving Forces of API Evolution
William Granli, John Burchell, Imed Hammouda, and Eric Knauss — *University of Gothenburg, Sweden; Chalmers University of Technology, Sweden* . . . 28

The Impact of Developer Team Sizes on the Structural Attributes of Software
Ahmmad Youssef and Andrea Capiluppi — *Brunel University, UK* . . . 38

Revisiting the Applicability of the Pareto Principle to Core Development Teams in Open Source Software Projects
Kazuhiro Yamashita, Shane McIntosh, Yasutaka Kamei, Ahmed E. Hassan, and Naoyasu Ubayashi — *Kyushu University, Japan; McGill University, Canada; Queen's University, Canada* . . . 46

Analysis Techniques

Software Evolution and Time Series Volatility: An Empirical Exploration
Jukka Ruohonen, Sami Hyrynsalmi, and Ville Leppänen — *University of Turku, Finland* . . . 56

Estimating Product Evolution Graph using Kolmogorov Complexity
Yasuhiro Hayase, Tetsuya Kanda, and Takashi Ishio — *University of Tsukuba, Japan; Osaka University, Japan* . . . 66

Using Control Flow Analysis to Improve the Effectiveness of Incremental Mutation Testing
Luke Bajada, Mark Micallef, and Christian Colombo — *University of Malta, Malta* . . . 73

Author Index

Localising Faults in Test Execution Traces

Gulsher Laghari Alessandro Murgia Serge Demeyer
ANSYMO
Universiteit Antwerpen — Middelheimlaan 1 — BE 2020 Antwerpen België
http://www.uantwerpen.be/en/rg/ansymo/
{gulsher.laghari,alessandro.murgia,serge.demeyer}@uantwerpen.be

ABSTRACT

With the advent of agile processes and their emphasis on continuous integration, automated tests became the prominent driver of the development process. When one of the thousands of tests fails, the corresponding fault should be localised as quickly as possible as development can only proceed when the fault is repaired. In this paper we propose a heuristic named SPEQTRA which mines the execution traces of a series of passing and failing tests, to localise the class which contains the fault. SPEQTRA produces ranking of classes that indicates the likelihood of classes to be at fault. We compare our spectrum based fault localisation heuristic with the state of the art (AMPLE) and demonstrate on a small yet representative case (*NanoXML*) that the ranking of classes proposed by SPEQTRA is significantly better than the one of AMPLE.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*debugging aids, tracing*

General Terms

Algorithms

Keywords

Automated developer tests; spectrum based fault localisation; replication (different heuristic & same data)

1. INTRODUCTION

The quintessential principle of continuous integration declares that software engineers should merge their working copies with the main branch several times a day [7]. During each integration step, a continuous integration server builds the entire project, using a fully automated process involving compilation, unit tests, integration tests, code analysis, security checks, etc. When one of these steps fails, the build is said to be *broken*; development can then only proceed when the fault is repaired [11, 15]. The safety net on automated tests, encourages software engineers to write lots of tests — several reports indicate that there is more test code than application code [16, 5, 19]. Moreover, executing all these tests sometimes take several hours [14]. Hence, it is critical to quickly identify the location of the fault in the code. Not only does a broken build block all progress in the team, but more importantly the location of the fault serves as an indicator for the software engineer expected to repair the build.

In the simplest case, there is a one-to-one mapping between the failing test and the class containing the fault. However, for complex object interactions where objects must adhere to a certain protocol, illegal call sequences trigger faults which are notoriously hard to pinpoint to an exact location [13]. Faults induced by illegal method call sequences are real and hard to debug: a conservative estimation identified 115 faults related to missing method calls in the Eclipse bug repository [12]. For such faults the one-to-one mapping between the failing test and the class containing the fault does not hold and then software engineers resort to debugging [21].

Luckily, there is a class of heuristics —named *spectrum based fault localisation*— which give indications for the location of the fault. Such heuristics compare execution traces of passing tests against the ones from a failing test, assuming that the points where the traces differ are the most likely location of the fault [1]. The state of the art heuristic for class level fault localisation by analysing method call sequences in unit tests is proposed by Dallmeier et al. with a tool called AMPLE [4]. AMPLE traces method calls invoked by objects and collects call sequences of the corresponding classes by sliding a window over the executions traces. It then compares the execution trace of all passing tests against one trace with a failing test and deduces which class most likely contains the fault. AMPLE was shown to be quite effective on a small yet representative case (*NanoXML*): it could immediately pinpoint the faulty class in 36% of all test runs; while on average 21% of the executed classes (10% of all classes) must be inspected to find the location of the fault.

In this paper we report on a replication experiment (different heuristic & same data) where we compare a new fault localisation heuristic named SPEQTRA against the state of the art AMPLE. SPEQTRA addresses two shortcomings of the AMPLE heuristic: (a) it filters out repetitive method calls (e.g. contained in loops) and (b) avoids the sliding window and the arbitrary upper limit on the length of the call sequence it imposes. To address these shortcomings, SPEQ-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.

IWPSE'15, August 30, 2015, Bergamo, Italy
ACM. 978-1-4503-3816-5/15/08...$15.00
http://dx.doi.org/10.1145/2804360.2804361

TRA uses a different algorithm (closed itemset mining) to characterise method call sequence and distinguish the faulty ones. Via the replication experiment we demonstrate that the ranking of classes proposed by SPEQTRA is significantly better than the one of AMPLE: we can immediately pinpoint the faulty class in 56% of all test runs; while on average 12% of the executed classes (5% of all classes) must be inspected to find the location of the fault. Moreover, for 70% of the faults, SPEQTRA has at most one false positive whereas for AMPLE this happens for 59% of the faults.

This paper is structured as follows. We explain the two heuristics under investigation in Section 2. Next, we describe the way we set up our replication experiment, including the particular details about the *NanoXML* case in Section 3. Then the bulk of the paper is contained within Section 4, where we show the results of the comparison including some anecdotal evidence from *NanoXML*. We list the related work, showing other research on dynamic analysis within a software evolution context in Section 5, followed by a discussion on the threats to validity in Section 6. Finally, Section 7 summarises our findings and lists the contributions.

2. HEURISTICS UNDER INVESTIGATION

In this section we give a detailed explanation of the two heuristics under investigation. First, we provide some background information regarding spectrum based fault localisation which is the basis for the two heuristics (Section 2.1). Then we contrast the two heuristics depicted in a Figure 1 showing where they are the same (i.e. collecting traces per object — Section 2.2) and where they differ (creating sequences of method calls — Section 2.3; the similarity coefficient used to rank the corresponding fault locations — Section 2.4).

2.1 Spectrum Based Fault Localisation

AMPLE and SPEQTRA both are instances of the class of spectrum based fault localisation heuristics [1]. Such heuristics discover statistical coincidences between system failures and the activity of the different parts of a system. All these heuristics create a so-called *program spectrum*, which is a matrix where each column corresponds to a program entity (e.g. statement, block, sequence of method calls) and rows represent a particular test run. For each test run the corresponding column for program entity is marked as 1 (executed) or 0 (not executed). Alongside the program spectrum, the heuristic also creates an *error vector* which is a column where a cell is marked as 1 if the test run failed or 0 if the test was a success. Next, the error vector is compared against all columns in the program spectrum using a particular *similarity coefficient*; the column which is most similar to the error vector is then the program entity which most likely contains the fault.

2.2 Collecting Traces

AMPLE and SPEQTRA use sequences of method calls as the program entities which are represented in each column of the fault spectrum matrix. Both heuristics group the outgoing method calls according to the following scheme. Let $O = \{o_1, o_2, ..., o_n\}$ be the set of object instances of the class C and $T = \{t_1, t_2, ..., t_n\}$ be the set of object traces of class C, where t_i represents the trace of outgoing method calls by the object o_i. By outgoing method calls we mean an object calling a method of another object. For instance if we have an object o_1 with a method $m1()$ and hit method hosts a call to method $n1()$ belonging to an object o_2, the collected outgoing method call for o_1 is $m1()$.

Two objects o_1 and o_2 of a class C may have following traces of method calls (Equations 1 and 2):

$$t_1 = \left\{ \begin{array}{c} m_1, m_1, m_1, m_2, m_2, m_3, \\ m_1, m_1, m_2, m_2, m_3 \end{array} \right\} \quad (1)$$

$$t_2 = \{m_1, m_1, m_1\} \quad (2)$$

AMPLE and SPEQTRA group all such object traces for the corresponding class C which has trace set T (Equation 3).

$$T = \left\{ \begin{array}{l} \{\ m_1, m_1, m_1, m_2, m_2, m_3, \\ m_1, m_1, m_2, m_2, m_3\}, \\ \{\ m_1, m_1, m_1\} \end{array} \right\} \quad (3)$$

2.3 From Traces to Class Sequences

Traces of outgoing method calls can grow to millions of method calls per object [3]. To reduce these traces AMPLE and SPEQTRA each apply a different technique to arrive at what we call *Class Sequences* for the remainder of the paper.

AMPLE — Sliding Window. AMPLE slides a window of fixed size over the trace to create a list of class sequences. From the previous example, if we fix the window size as 2 and slide it over the object traces in Equation 3 we obtain the set of class sequences in Equation 4

$$C_A = \left\{ \begin{array}{l} \{\{m_1, m_1\}, \{m_1, m_2\}, \{m_2, m_2\}, \\ \{m_2, m_3\}, \{m_3, m_1\}\} \end{array} \right\} \quad (4)$$

SPEQTRA — Frequent Sequences. To avoid the arbitrary upper limit imposed by the size of the sliding window, SPEQTRA incorporates the frequently appearing sequences adopting an algorithm named *closed itemset mining* [20]. Given the set of object traces T of class C, we define:

- X —*itemset*— a set of method calls.
- $\sigma(X)$ —support of X— the number of traces of T that contain this itemset X.
- *minsup* —minimum support of X— a threshold used to tune the number of returned itemsets.
- *frequent itemset* — an itemset X is frequent when $\sigma(X) \geq minsup$.
- *closed itemset* — a frequent itemset X is closed if there exists no proper superset X' whose support is same as the support of X (i-e. $\sigma(X') = \sigma(X)$).

From now on, we refer a closed itemset X as a frequent sequence or simply a sequence of method calls. Adopting closed itemset mining in the context of fault localisation, we fix *minsup* to 1 because those classes which only create one object (and thus one trace) should be included in the program spectrum as well; this one call trace may be the one which triggers the fault. However, we tune the algorithm in another way. The mining algorithm also returns frequent sequences that comprise only one method call. Since we are looking for faults caused by complex object interactions where objects must adhere to a certain protocol, sequences should have at least a length of two.

From the previous example with input T (Equation 3) and $minsup = 1$ the generated set of frequent sequences is:

$$C_F = \{\{m_1\}, \{m_1, m_3, m_2\}\} \quad (5)$$

It can be observed that the sequences such as $\{m_2\}$, $\{m_3\}$, $\{m_1, m_2\}$, $\{m_1, m_3\}$, $\{m_2, m_3\}$ are not included in final set (Equation 5), since there exists a super sequence $\{m_1, m_3, m_2\}$

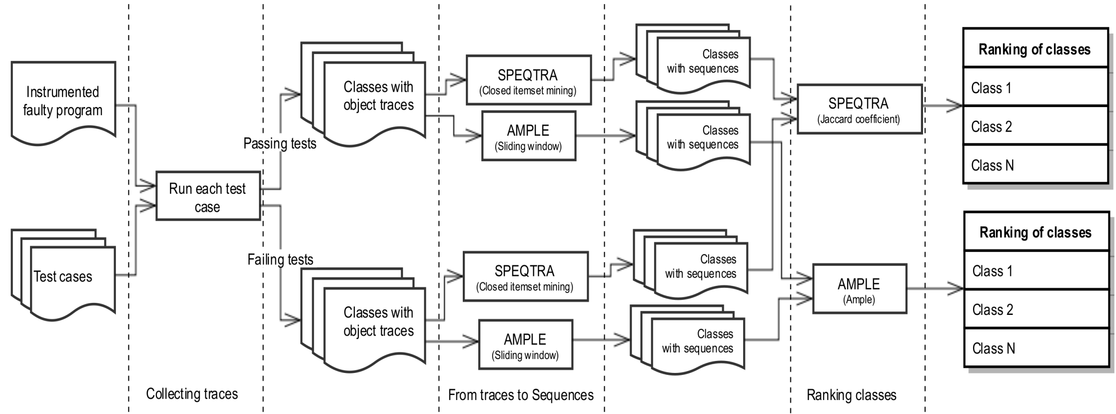


Figure 1: Overview of the two heuristics showing three steps (i) Collecting tracing, (ii) Collecting class sequences, and (iii) Ranking classes

with equal support. As SPEQTRA removes all frequent sequences of length 1, thus the sequence set C_F (5) is finally reduced into the set of Class Sequences C_S in (6).

$$C_S = \{\{m_1, m_3, m_2\}\} \tag{6}$$

2.4 Ranking Classes

Both AMPLE and SPEQTRA assign a weight $W(X)$ to each class sequence in C_A and C_S(Equation 4 and 6 respectively). Note that X refers to a sequence of method calls, but there is a difference in that X is a chunk of fixed size in AMPLE whereas X is set of frequent method calls in SPEQTRA.

AMPLE — AMPLE has defined its own weighting scheme based on a configuration of a single failing test and several passing tests. Sequences in AMPLE are assigned a weight between 0 and 1 using equation (7) [4].

$$W(X) = \begin{cases} \frac{k(X)}{n} & \text{if } X \text{ not in failing test} \\ 1 - \frac{k(X)}{n} & \text{if } X \text{ in failing test} \end{cases} \tag{7}$$

Where n is the number of passing tests and $k(X)$ is the number of passing tests that include the sequence X.

SPEQTRA — We tested several weighting schemes to rank the classes. Ultimately, in SPEQTRA, we opted for the Jaccard similarity coefficient (Equation 8) adopted from Chen et al. [2]:

$$W(X) = \frac{a_{11}(X)}{a_{11}(X) + a_{01}(X) + a_{10}(X)} \tag{8}$$

Where:

- $a_{11}(X)$ = Number of failing tests in which *sequenceX* is found.
- $a_{10}(X)$ = Number of passing tests in which *sequenceX* is found.
- $a_{01}(X)$ = Number of failing tests in which *sequenceX* is not found.

Weight per class. Both AMPLE and SPEQTRA take the average of all weights for all sequences of a class and assign this weight to class C, as defined in Equation 9:

$$W(C) = \frac{1}{n}\sum_{i=1}^{n} W(X_i) \tag{9}$$

where n is the number of sequences in the class and $W(X_i)$ is weight of a sequence as given by Equation 7 (AMPLE) or Equation 8 (SPEQTRA).

Finally, both heuristics rank all classes using their weights $W(C)$, where the one with the highest weight is the most likely location of the fault.

3. EXPERIMENTAL SETUP

This paper is set-up as a replication experiment (different heuristic & same data) where we compare a new fault localisation heuristic named SPEQTRA against the state of the art AMPLE. In what follows, we describe the way we set up our replication experiment, including the particular details about the *NanoXML* case (subsection 3.1). Next, we provide the necessary practical details about the mechanics of the experiment so that other researcher can replicate our findings (subsection 3.2)

3.1 Replication Case — NanoXML

The original paper proposing the AMPLE heuristic demonstrated its effectiveness on a small but representative project named *NanoXML* [4]. *NanoXML* is a non-validating XML parser written in Java. Its source code and documentation are available in the Software-artifact Infrastructure Repository[1] [6].

NanoXML has five development versions (V1 ... V5) where the number of classes span from 16 to 23 (Table 1). With the exception of version V4, all others have documented faults that can be activated and exposed by the test suite. These versions (V1,V2,V3,V5) —the ones we use for the experiment— have 32 faults (cumulatively). Each version is

[1] http://sir.unl.edu/portal/index.php

Table 1: *NanoXML* version details

Version	# of classes	LOC	# of faults	# of tests
1	16	4334	7	214
2	19	5806	7	214
3	21	7185	10	216
5	23	7646	8	216

shipped along with tests and test drivers. A test driver is a class that sets up multiple tests by feeding them with the required input (e.g. read a XML file). The goal of these test drivers is to trigger one and only one feature of the project.

We collect the traces of all faults by injecting one fault at a time and subsequently running the test suite. Then for each test, we record the outgoing methods calls of the created objects. It is important to note that, for each test (passing or failing), a separate trace is maintained for each object. All the outgoing method calls of an object appear in their own trace. Thus, two objects o_1 and o_2 of the same class C have independent traces of method calls.

In the replication experiment we activate one fault at a time. Having 32 faults leads to 32 distinct variants of *NanoXML*. Among these variants we picked only the ones that generate at least one failing and one passing test for each test driver. Just like in the original AMPLE experiment, this ensures that failing and passing tests are all related to the same functionality. For each test driver, we group one failing test with all passing tests, the set-up that is needed to reflect the set-up of AMPLE experiment. We repeat this process for each failing test associated to that test driver. At the end of this process, we end up with 18 variants of *NanoXML* and 347 combinations of failing and passing tests used for our experiments.

Note that this set-up is not exactly the same as the one reported in the AMPLE paper because the version of *NanoXML* we downloaded from the Software-artifact Infrastructure Repository has been changed. In the latest version, one fault is removed from V5 with a note "since it is overly expressive it may not be representative of a *pseudo-real* fault". As a consequence, the fault matrix also differs from the previous version. This is an inherent risk with replication experiments and partially explains why we do not obtain the same results reported in the AMPLE experiment [4].

3.2 Replication Details

AMPLE replication.

When preparing for the replication experiment, we downloaded the original binary of the AMPLE implementation. Due to hardware constraints, we were unable to run this binary. Consequently, we implemented our own version of the algorithm as reported in the original AMPLE paper [4]. We used the optimal settings for the parameters of the heuristic, in particular we adopted a sliding window size of 8.

AspectJ. The object traces are collected by introducing logger functionality into the *NanoXML* code via AspectJ[2]. More specifically, we use a method call join point with a pointcut to pick out every call site. Each time a method call occurs, the aspect extracts the caller object and adds a method entry to the object's trace. The aspect is robust for different threads that may be running within the java project, although this was irrelevant for the *NanoXML* case. All object traces belonging to same class appear together in a HashMap maintained for each executed class.

Static Methods are Ignored. SPEQTRA, like AMPLE, also collects traces of method calls invoked by objects of a class. Calls to static methods are not captured and do not appear in the trace, hence cannot be identified as the location of the fault. For the particular replication of the *NanoXML* experiment this did not cause any problems however this limitation must be taken into account for future replication.

Single failing test. For this experiment, we inject one fault into the program which causes one or more tests to fail and several of them to pass. All these tests are executed with same test driver. In principle, SPEQTRA is able to rank fault locations using all these failing and related passing tests. However, since AMPLE is designed to work with only one failing test we replicated the set-up to include the trace of a single failing test and one or more passing tests.

Closed Itemset Mining. To avoid the arbitrary upper limit imposed by the size of the sliding window, SPEQTRA incorporates the frequently appearing sequences adopting an algorithm named *closed itemset mining* [20]. In particular, we used the implementation provided by the library SPMF[3].

Search Length. To compare the results of the two heuristics we use the so-called *search length* as defined in the AMPLE experiment [4]. The search length counts how many classes are placed atop of the faulty class in the ranking produced by the heuristic. In that sense, it represents how many classes the developer has to examine before finding the class containing the fault. The search length is zero whenever the faulty class is placed as the first item in the ranking.

4. RESULTS AND DISCUSSION

To compare our results with AMPLE, we replicated AMPLE with a sliding window size 8, which is the value for which AMPLE achieved the best performance. Following the experimental set-up explained in Section 3, we obtained rankings for all the 347 combinations both with our implementation of AMPLE and SPEQTRA. Below we discuss the results of both.

1. Our replication experiment confirmed the results reported in the original AMPLE paper. There were some minor changes in the results, but these can be attributed to the differences in the *NanoXML* version used in our experiment, in particular in the tests accompanying the project.
2. The average search length of all rankings in 347 test runs with both heuristics is reported in Table 2. Here SPEQTRA has less average search length than sliding window approach.
3. In 173 test runs out of 347, SPEQTRA outperforms AMPLE and has search length less than AMPLE. In 140 test runs both SPEQTRA and AMPLE have same search length, whereas in only 34 cases SPEQTRA has search length greater than AMPLE.
4. The plot of the cumulative of search length distribution in 347 test runs with both heuristics is given in Figure 2. With SPEQTRA, the search length of 0 covers 56% of faults, whereas with AMPLE it is only 40% of faults. Furthermore, the worst case search length with SPEQTRA is 6 whereas with AMPLE it is 8.

[2]AspectJ http://eclipse.org/aspectj/

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

Table 2: Average Search Length in SPEQTRA and AMPLE

AMPLE	SPEQTRA
2.07	1.20

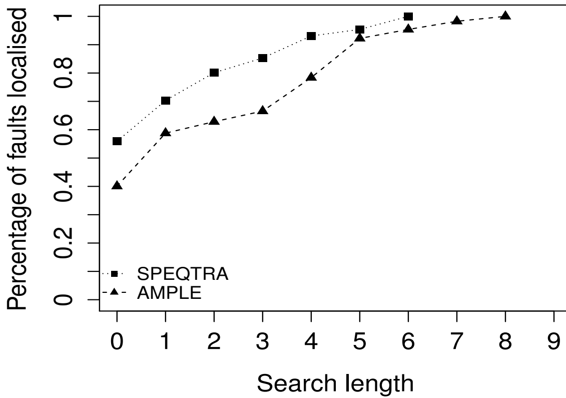


Figure 2: Search Length in SPEQTRA and AMPLE.

4.1 Anecdotal Evidence

From the experiment we collected some anecdotal evidence highlighting the main differences between the two approaches. Specifically the two differences, namely (i) the effect of the sliding window and (ii) the impact of repetitive method calls (e.g. contained in loops). In Listing 1, we see a piece of source code showing a fault in `XMLElement` class at line 7 which was exercised by several unit tests. This resulted in three `XMLElement` object traces generated by the failing test as shown in Listings 2, 3 and 4. Note that for brevity, method parameters are not shown.

From these object traces in failing test, SPEQTRA generated three sequences of method calls for class XMLElement. For brevity, we list the numbers in the sequences instead of method calls. These numbers in the sequences represents the line numbers in Listing 2, unless otherwise mentioned. The line number for the method is the very first entry of the method in the trace. The first sequence was s1 = {1, 4, 21, 24, 5, 15, 7}, the methods indicated by numbers 5 and 7 appear in Listing 3 at lines 5 and 7. The second sequence was s2 ={1, 4, 13, 21, 24, 15} and the third sequence was s3 = {1, 4, 21, 24, 15}. These sequences capture frequent method calls occurring in the traces and hence represent a good abstraction of the traces. The three sequences have length 7, 6 and 5 respectively and none of the sequence has repetitive method calls.

On the other hand, AMPLE generated 37 sequences for the class, each with repetitive method calls. If we slide a window of size 8 over the trace in Listing 2, the first sequence s1 = {1, 2, 3, 4 ,5, 6, 7, 8} contains 5 repetitions for method `XMLElement.findAttribute()` and three repetitions of method `XMLAttribute.getFullName()`. Likewise the second sequence s2 = {2, 3, 4 ,5, 6, 7, 8, 9} contains four times method `XMLElement.findAttribute()` and 4 times method `XMLAttribute.getFullName()`. As a consequence, the AMPLE heuristic fails to locate the fault and it ranked the faulty class on position 6 whereas SPEQTRA could pinpoint it exactly.

Listing 1: Code snippet with a fault in XMLElement class

```
1  public Enumeration
       enumerateAttributeNames() {
2   Vector result = new Vector();
3   Enumeration _enum = this.attributes.
        elements();
4   while (_enum.hasMoreElements()) {
5    XMLAttribute attr = (XMLAttribute)
         _enum.nextElement();
6    // The call should be to attr.getName()
7    result.addElement(attr.getFullName());
8   }
9   return result.elements();
10 }
```

Listing 2: Failing trace of XMLElement object 1

```
1  XMLElement.findAttribute(String)
2  XMLElement.findAttribute(String)
3  XMLElement.findAttribute(String)
4  XMLAttribute.getFullName()
5  XMLElement.findAttribute(String)
6  XMLAttribute.getFullName()
7  XMLElement.findAttribute(String)
8  XMLAttribute.getFullName()
9  XMLAttribute.getFullName()
10 XMLElement.findAttribute(String)
11 XMLAttribute.getFullName()
12 XMLAttribute.getFullName()
13 XMLElement.getName()
14 XMLElement.getName()
15 XMLAttribute.getName()
16 XMLAttribute.getName()
17 XMLAttribute.getName()
18 XMLAttribute.getName()
19 XMLAttribute.getName()
20 XMLAttribute.getName()
21 XMLElement.getAttribute()
22 XMLElement.findAttribute(String)
23 XMLAttribute.getFullName()
24 XMLAttribute.getValue()
25 XMLElement.getAttribute()
26 XMLElement.findAttribute(String)
27 XMLAttribute.getFullName()
28 XMLAttribute.getFullName()
29 XMLAttribute.getValue()
30 XMLElement.getAttribute()
31 XMLElement.findAttribute(String)
32 XMLAttribute.getFullName()
33 XMLAttribute.getFullName()
34 XMLAttribute.getFullName()
35 XMLAttribute.getValue()
```

There are however 34 situations where AMPLE was more accurate than SPEQTRA. We use one fault injected in the class `NonValidator` to explain this difference. The failing test and several of the passing tests generated the same trace for the only object of `NonValidator`. In this case, SPEQTRA generated one sequence for the failing test and two sequences for the passing tests, one of which also appeared in the failing test. As a consequence, SPEQTRA ranked this class on position 4 whereas AMPLE could pinpoint it exactly. This can be explained as, the Jaccard similarity coefficient (or any other coefficient used in spectrum based fault localisation) assigns more weight to a sequence when it is present more in failing tests and less in passing tests. Hence the sequence only present in passing tests gets weight 0; the value 0 for numerator $a_{11}(X)$ in equation 8 evaluates the whole equation to 0. The other sequence presented in failing test gets a lower weight due to its presence in several passing tests; the higher

value of denominator $a_{10}(X)$ in equation 8 decreases the value. Consequently, the weight of the class, which is average weight of the two sequences, is also less.

Listing 3: Failing trace of XMLElement object 2

```
1  XMLElement.findAttribute(String)
2  XMLElement.findAttribute(String)
3  XMLElement.findAttribute(String)
4  XMLAttribute.getFullName()
5  XMLElement.findAttribute(String, String)
6  XMLAttribute.getName()
7  XMLAttribute.getNamespace()
8  XMLAttribute.getName()
9  XMLAttribute.getName()
10 XMLAttribute.getName()
11 XMLAttribute.getName()
12 XMLElement.getAttribute( )
13 XMLElement.findAttribute(String)
14 XMLAttribute.getFullName()
15 XMLAttribute.getValue()
16 XMLElement.getAttribute( )
17 XMLElement.findAttribute(String)
18 XMLAttribute.getFullName()
19 XMLAttribute.getValue()
```

Listing 4: Failing trace of XMLElement object 3

```
1  XMLElement.getName()
```

4.2 Discussion

Based on this replication experiment, we conclude that SPEQTRA is significantly better than AMPLE since:

1. The average ranking in SPEQTRA is lower than AMPLE. This average suggests that a developer has to search through, on average, 12% of 10.25 average executed classes or 5% of all 23 classes. This is significantly better than AMPLE, where 20% of executed classes or 9% of all classes need to be searched.
2. A faulty class is placed first in the ranking (search length 0) for 56% of faults by SPEQTRA whereas for AMPLE it happens only for 40% of the faults.
3. For 70% of faults, there is atmost one false positive (search length 1) with SPEQTRA whereas for AMPLE this happens for 59% of the faults.

5. RELATED WORK

In this section, we present related work on spectrum based fault localisation thus immediately relevant for this replication experiment. Moreover, we also give references to related work on dynamic analysis for program comprehension as this provides the broader context for our research.

5.1 Spectrum Based Fault Localisation

Spectrum based fault localisation is an automated fault diagnosis technique based on differences in program spectra of a program between passing and failing tests [1]. Spectrum based fault localisation techniques have been applied in many domains such as localising the fault and ranking program statements [10], blocks [1], failure related components [2] and —last but not least— classes [4].

- Jones et al. used statement-hit spectra to rank statements of C programs according to their likelihood to be at fault [10]. To visualize the ranking, they implemented a tool — Tarantula — able to mark statements with colors that span from red (statement likely at fault) to green (statement unlikely at fault).
- Abreu et al. used blocks-hit spectra to rank the blocks in order of their likelihood to be at fault [1]. Here the block is defined as C language statement where compound statement (statements inside curly brackets) counts as a single statement. They compared the performance of different similarity coefficients and their impact on diagnostic accuracy of spectrum based fault localisation technique.
- Chen et al. implemented the tool Pinpoint for tracing client requests in Internet service environments. Pinpoint records the components involved in the service and whether or not the request is satisfied [2]. The tool correlates the request failures to the components that most likely caused the failure.

All previous papers detect the fault at different levels of granularity(e.g., statements, blocks). However, none of them uses spectrum based fault localisation to identify faults due to method call sequences. In the literature, the only two techniques able to localise faults related to method call sequence are AMPLE and MCA-E. AMPLE is a tool, created by Dallmeier et al., that traces method calls invoked by objects and collects call sequences of the corresponding classes [4]. The outcome of the tool is a list of classes ranked according to their likelihood to be at fault. MCA-E is a technique proposed by Tu et al. in order to improve the regular spectrum based fault localisation techniques adopted in AMPLE [17]. Its outcome is a list of statements ranked according to their likelihood to be at fault. In the first step, it computes the likelihood of classes to be at fault (suspiciousness) by taking into account the difference of their method call sequences between passing and failing tests. In the second step, the suspiciousness of classes is used together with their statements to generate ranking of statements. One of the limitations of AMPLE and MCA-E approaches is the adoption of a window of finite size that slides over the execution traces. Such sliding a window is not efficient for computing the sequences since method calls may stem from loops or may repeat in a trace resulting into sequences with repetitive method calls which add overhead with little extra information. Furthermore, the number of sequences linearly increases as the size of the trace increases. With window size w and n number of method calls in a trace, n^w sequences are possible. In this paper we address the shortcomings related to the sliding window by mining the frequent method call sequences. The sequence mining alleviates the arbitrary upper limit on the length of the call sequence. It also optimises the computational power required to obtain the ranking of classes as the mining algorithm limits the frequent sequences to closed ones: and also the SPEQTRA removes one-length sequences, the number of SPEQTRA sequences is far less than sliding a window over the trace.

It also optimises the computational power required to obtain the ranking of classes as it generates far less sequences than sliding a window over the trace.

5.2 Program Comprehension

Discovering program invariants and specifications such as legal method call sequences are common goals of research in program comprehension [8, 9, 13]. Such specifications, achieved by means of dynamic analysis, are used for purposes including documentation, learning the API's etc.

Ernst et al. implemented the tool Daikon to dynamically detect program invariants [8]. An invariant is defined as a property that is true at a particular program point or points. Daikon runs an instrumented program over a series of test runs and records program properties. At the invariant detection stage it starts with a list of hypothetical invariants comparing them across all the traced properties of the program for all test runs. It immediately discards the hypothetical invariant the moment it does not hold for a test run. Finally, all the invariants that are validated across all test runs are reported. Daikon can also be used to detect invariant violations in failing tests and as such may be used in a similar set-up as what we report here.

Gabel et al. implemented OCD, a tool which traces method calls and, using a predefined template as a model for specification inference, learns and enforces temporal specifications over method call sequence [9]. The algorithm suffers from two limitations: (1) the template limits the sequence to comprise only two method calls and (2) the sequences inferred from a limited window size. The efficiency of the algorithm critically depends on the window size. Experimenting with Eclipse and Ant, the tool detected a few anomalies as violations of inferred sequences, though the anomalies did not result in program crashes.

Pradel et al. proposed a dynamic analysis technique to infer specifications of correct method call sequences [13]. The technique focuses on object collaboration, namely objects and method calls used together in the execution of a single method. By running a software program, the technique traces method calls, computes object collaborations and identifies patterns among these collaborations. From these patterns, the technique infers the legal method call sequence in the form of finite state machines.

To certain extent, our research on SPEQTRA is complementary to the previous ones. We use method call sequences of a class from passing tests, which can be assumed as usage patterns of the program. On the other hand, the sequences in failing tests can be considered as deviant behaviour.

6. THREATS TO VALIDITY

Following the template for case studies in [18], we discuss the threats to validity that can affect our results.

Threats to **external validity** correspond to the generalizability of our experimental results. Our study is limited to the object oriented system *NanoXML*. Although *NanoXML* is small project, it represents a good testbed since it provides documented tests and faults for replicating our study. Moreover, by using the same case study of Dallmeier et al. [4], we were able to verify the impact of removing the sliding window and adopting different similarity coefficients for mining faults in stack traces. Nevertheless, it is desirable to replicate our findings using other projects.

Threats to **internal validity** concern confounding factors that can influence the obtained results. Our approach leverages on the fault's "ability" of changing the stack trace generated by software execution. In that sense, we localise faults by pointing out the class that has different method call sequences (in passing and failing tests) assuming that such deviation is due to the fault. This assumption is a key-element in spectrum based fault localisation techniques based on method call sequences [4, 17] and our results confirm its general validity. On the other hand, there are cases where it does not apply. In *NanoXML* there is (only) one faulty class that cannot be localised —with our approach— since the fault is caused by a variable accessed without any method call.

Threats to **construct validity** focus on how accurately the observations describe the phenomena of interest. Our experiment relies on the correct identification of fault responsible for test failure. From this point of view, we do not have threats to construct validity since we inject one fault at the time. When the fault is injected, otherwise the test passes.

Threats to **reliability validity** correspond to the degree to which the same data would lead to the same results when repeated. We describe all steps of our technique and provide references on any tool or library involved in the analysis. The case study we use is publicly available in the Software-artifact Infrastructure Repository[4], a repository created for supporting rigorous controlled experimentation with program analysis and software testing techniques [6].

7. CONCLUSION

In this paper we presented a novel spectrum based fault localisation heuristic (named SPEQTRA) which used closed itemset mining to identify the characteristic method call sequences and the Jaccard similarity coefficient to rank the classes according to the likelihood of containing the fault. We compare our fault localisation heuristic with the state of the art (AMPLE) and demonstrate on a small yet representative case (*NanoXML*) that the ranking of classes proposed by SPEQTRA is significantly better than the one of AMPLE. In particular, SPEQTRA can immediately pinpoint the faulty class in 56% of all test runs (against 40% for AMPLE); while on average 12% of the executed classes must be inspected to find the location of the fault (against 20% for AMPLE). From anecdotal evidence, we deduce that the main reason why SPEQTRA performs better than AMPLE is due to closed itemset mining: this filters out repetitive method calls (e.g. contained in loops) and avoids the arbitrary upper limit imposed by the sliding window. Nevertheless, for a few faults AMPLE provides a better ranking than SPEQTRA, caused by call sequences appearing in both failing and many passing tests which reduced the weight of sequences.

Over the course of this research, we have made the following contributions:

- *Replication Experiment.* We conducted a replication of the AMPLE experiment performed by Dallmeier et al. [4]. We used the same data (the *NanoXML* case provided in the Software-artifact Infrastructure Repository [6]) and confirmed the numbers provided in the original report.
- *Alternative Heuristic.* We proposed an alternative spectrum based fault localisation heuristic (named SPEQTRA). We compared it against the results from AMPLE. We demonstrate that the ranking of classes proposed by SPEQTRA is significantly better than the one of AMPLE.
- *Anecdotal Evidence.* We collected some anecdotal evidence from the *NanoXML* case interpreting the main differences between the two heuristics.

Fault localisation heuristics are particularly relevant in modern software engineering owing to the increasing popularity of continuous integration. Continuous integration states that software engineers should merge their working copies

[4]http://sir.unl.edu/portal/index.php

with the main branch several times a day using a suite of automated tests to verify the correctness of the build. When one of the thousands of tests fails, the corresponding fault should be localised as quickly as possible as development can only proceed when the fault is repaired. In that sense our work shows that while the state of the art is rapidly advancing, it is worthwhile to make improvements on research from a decade ago.

8. ACKNOWLEDGMENTS

This work is sponsored by (i) the Higher Education Comission of Pakistan under a project titled "Strengthening of Univeristy of Sindh (Faculty Development Program)"; (ii) the Institute for the Promotion of Innovation through Science and Technology in Flanders through a project entitled "Change-centric Quality Assurance (CHAQ)" with number 120028.

9. REFERENCES

[1] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780–1792, Nov. 2009.

[2] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, DSN '02, pages 595–604, Washington, DC, USA, 2002. IEEE Computer Society.

[3] B. Cornelissen, A. Zaidman, D. Holten, L. Moonen, A. van Deursen, and J. J. van Wijk. Execution trace analysis through massive sequence and circular bundle views. *Journal of Systems and Software*, 81(12):2252 – 2268, 2008.

[4] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight defect localization for java. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, ECOOP'05, pages 528–550, Berlin, Heidelberg, 2005. Springer-Verlag.

[5] B. Daniel, V. Jagannath, D. Dig, and D. Marinov. Reassert: Suggesting repairs for broken unit tests. In *Proceedings of the Int'l Conference on Automated Software Engineering (ASE)*, pages 433–444. IEEE CS, 2009.

[6] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

[7] P. M. Duvall, S. Matyas, and A. Glover. *Continuous Integration: Improving Software Quality and Reducing Risk.* Addison-Wesley, 2007.

[8] M. D. Ernst, J. H. Perkins, P. J. Guo, S. Mccamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. In *Science of Computer Programming*, 2006.

[9] M. Gabel and Z. Su. Online inference and enforcement of temporal properties. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 15–24, New York, NY, USA, 2010. ACM.

[10] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 273–282, New York, NY, USA, 2005. ACM.

[11] A. Miller. A hundred days of continuous integration. In *Agile, 2008. AGILE '08. Conference*, pages 289–293, Aug 2008.

[12] M. Monperrus and M. Mezini. Detecting missing method calls as violations of the majority rule. *ACM Trans. Softw. Eng. Methodol.*, 22(1):7:1–7:25, Mar. 2013.

[13] M. Pradel and T. R. Gross. Automatic generation of object usage specifications from large method traces. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 371–382, Washington, DC, USA, 2009. IEEE Computer Society.

[14] P. Runeson. A survey of unit testing practices. *IEEE Software*, 23(4):22–29, 2006.

[15] D. Ståhl and J. Bosch. Modeling continuous integration practice differences in industry software development. *Journal of Systems and Software*, 87(0):48 — 59, 2014.

[16] N. Tillmann and W. Schulte. Unit tests reloaded: Parameterized unit testing with symbolic execution. *IEEE Software*, 23(4), 2006.

[17] J. Tu, L. Chen, Y. Zhou, J. Zhao, and B. Xu. Leveraging method call anomalies to improve the effectiveness of spectrum-based fault localization techniques for object-oriented programs. In *Quality Software (QSIC), 2012 12th International Conference on*, pages 1–8, Aug 2012.

[18] R. K. Yin. *Case study research: Design and methods.* Sage publications, 2013.

[19] A. Zaidman, B. V. Rompaey, van Arie van Deursen, and S. Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*, 16(3):325–364, 2011.

[20] M. J. Zaki and C. J. Hsiao. CHARM: an efficient algorithm for closed itemset mining. In *Proceedings of the Second SIAM International Conference on Data Mining, Arlington, VA, USA, April 11-13, 2002*, pages 457–473, 2002.

[21] A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging.* Morgan Kaufmann, 2005.

Circumventing Refactoring Masking using Fine-Grained Change Recording

Quinten David Soetens, Javier Pérez and Serge Demeyer
University of Antwerp
Antwerpen, Belgium
{quinten.soetens; javier.perez; serge.demeyer}@uantwerp.be

Andy Zaidman
Delft University of Technology
Delft, The Netherlands
a.e.zaidman@tudelft.nl

ABSTRACT

Today, refactoring reconstruction techniques are snapshot-based: they compare two revisions from a source code management system and calculate the shortest path of edit operations to go from the one to the other. An inherent risk with snapshot-based approaches is that a refactoring may be concealed by later edit operations acting on the same source code entity, a phenomenon we call *refactoring masking*. In this paper, we performed an experiment to find out at which point refactoring masking occurs and confirmed that a snapshot-based technique misses refactorings when several edit operations are performed on the same source code entity. We present a way of reconstructing refactorings using fine grained changes that are recorded live from an integrated development environment and demonstrate on two cases —PMD and Cruisecontrol— that our approach is more accurate in a significant number of situations than the state-of-the-art snapshot-based technique RefFinder.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

Keywords

Refactoring Reconstruction; Refactoring Masking; Fine Grained Changes; Software Evolution

1. INTRODUCTION

Refactoring is widely recognised as a crucial technique applied when evolving object-oriented software systems. The key idea is to redistribute program entities and responsibilities in order to prepare the software for future extensions. If applied well, refactoring improves the design of software, makes software easier to understand, helps to find bugs, and helps to program faster [10]. As such, refactoring has received widespread attention within both academic and industrial circles, and is mentioned as a recommended practice in the software engineering body of knowledge [1].

Given this widespread attention, several researchers set out to reconstruct refactorings as they occurred in the evolution of software projects. Initially, this was mainly an act of scientific curiosity (*i.e.*, [3, 21, 37, 39, 27, 14]), however later on actual applications emerged. Weißgerber *et al.* for instance used this as a means for studying the impact of refactorings on defects [11, 38]. Dig *et al.* prototyped a capture-playback tool capable of replaying refactorings when migrating systems dependent on a refactored API [13, 7, 8]. Obviously, several authors tried to correlate the impact of refactorings on the maintainability of a software project [32, 16, 22, 36].

In the meantime, several field studies and surveys indicated that if refactoring is applied in practice, it is mainly interwoven with normal software development [17, 15]. A side effect of this interweaving is that a commit in a source code management system tends to consist of more than just a single refactoring [2]. Indeed Negara *et al.* reported that 46% of refactored program entities are also edited in the same commit [19]. Consequently, state of the art refactoring reconstruction techniques miss a significant portion of the actual refactorings, because they infer refactorings by comparing two revisions of a system and making educated guesses about the precise edit operations applied in between. At that point, it is virtually impossible for such snapshot-based refactoring reconstruction tools to correctly deduce refactorings since these may be concealed by other changes. Negara *et al.* found that on average 30% of refactoring operations do not reach the version control system [18]. We call this the *"refactoring masking"* phenomenon and investigate the nature of the problem in a first Research Question.

RQ 1 – Refactoring Masking. *Under which conditions does a snapshot-based approach fail to reconstruct refactorings?*

To address this first research question we followed the refactoring script of a small yet representative program (the LAN Simulation [6]) and committed individual atomic changes to separate revisions in a source code repository. We then ran RefFinder [14] to compare all possible combinations of revisions to investigate under which conditions RefFinder fails to reconstruct the refactorings. We found that some refactorings indeed conceal others as they act on the same source code entities. Combinations of ExtractMethod and MoveMethod are particularly vulnerable.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

IWPSE'15, August 30, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3816-5/15/08...$15.00
http://dx.doi.org/10.1145/2804360.2804362

A solution to this problem might be to use the actual changes, as performed in an integrated development environment. Assuming that an integrated development environment provides logging facilities for all editing operations (*i.e.*, like done in Spyware [25], Syde [12], Cheops [9], OperationRecorder [20] and ChEOPSJ [33]), we might query this stream of changes to distinguish refactorings from ordinary program edits. We performed a proof by construction via a tool prototype named ChEOPSJ, which sits in the background of Eclipse and records the changes made to a software system while a developer is programming. We compared this tool prototype against the state of the art snapshot-based approach as explained by the second research question.

RQ 2 – Comparison. *Do fine-grained changes allow us to reconstruct refactorings where snapshot-based approaches fail?*

We compare the change-based approach (exemplified by ChEOPSJ [33]) against a snapshot-based approach (exemplified by RefFinder [14]) on two open source cases – PMD and Cruisecontrol. We locate instances of the refactoring masking phenomenon and show that the change-based approach is indeed more accurate in reconstructing refactorings in those cases. Moreover, we argue that this improved accuracy is relevant by estimating the number of edit operations acting on the same source code entities within 5 minutes after an ExtractMethod refactoring.

We structured the remainder of this paper as follows. Section 2 introduces the state of the art, including a description of the ChEOPSJ tool prototype. Next we have two sections 3, and 4, which each address one of the research questions with their own experimental setup and results. The final two sections 5 and 6 wrap up the paper with a discussion of the limitations of – and threats to – the validity of our research and summarise our major findings in the conclusions.

2. STATE OF THE ART

2.1 Snapshot-Based Reconstruction

In this paper, we use the term *"refactoring reconstruction"* to refer to any software reengineering technique used to infer refactorings that were performed in the history of a software system. The current state of the art in this consists of analyses of snapshots maintained in a source code repository. Most approaches use some kind of code similarity measure to identify possible refactoring candidates. Dig *et al.* as well as Weißgerber *et al.* used a combination of a signature-based analysis and shingles (a form of hashing) [7, 8, 11, 38]. Van Rysselberghe *et al.* use clone detection on two versions to look for a decrease in the number of clones. Since many refactorings are aimed at the elimination of duplicated code [37], this would suggest that a refactoring was performed. There exist two approaches that do not rely on code similarity. Demeyer *et al.* developed a set of heuristics to identify refactorings using decreasing code entities [3]. Xing and Stroulia search for refactorings at the design level using their UMLDiff algorithm, which is capable of detecting some basic structural changes to the system [39] [27].

RefFinder, an Eclipse plugin by Kim *et al.* is to date the most comprehensive refactoring reconstruction tool as it supports 63 different types of refactorings [14]. They use the technique proposed by Prete *et al.*, which is stronger than all previous techniques because they not only detect primitive refactorings (which all previous techniques do to some extent)

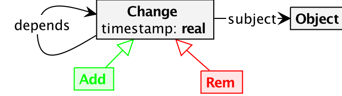


Figure 1: The types of changes implemented.

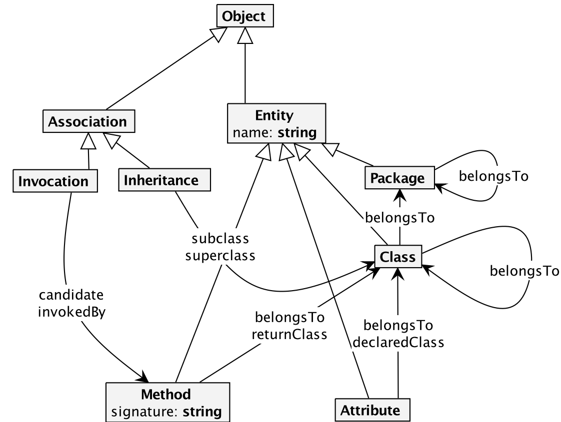


Figure 2: The types of source code entities implemented. (note: For the sake of readability, when multiple relationships exist between entities, a single arrow is drawn.)

but also "complex refactorings" (*i.e.*, refactorings which are combinations of primitive refactorings). To do this they rely on a fact base with a strong query engine (Tyruba logic) [21]. They describe the structural constraints before and after applying a refactoring in terms of template logic queries. RefFinder takes two versions of a system as input from the Eclipse workspace and recovers changes as logic facts about the systems' syntactic structure using LSDiff. These are then stored in a factbase, which can be queried to identify program differences that match the constraints of each refactoring type under focus.

We opted to use RefFinder in our experiments, because it is a representative of the state-of-the art in snapshot-based refactoring reconstruction techniques and because the list of refactorings it is able to detect is currently the most comprehensive.

2.2 Change-Based Reconstruction

An alternative to the snapshot-based approach is to use the actual edit operations as performed in an integrated development environment. In such an approach a tool silently records the activities of the programmers while they are working, and registers all the changes as performed. For instance, if the programmer modifies a method, the recorder instantiates change objects for each of the statements that were added, changed or removed. This approach was used by Robbes and Lanza in Spyware [25]; by Hattori and Lanza in Syde [12]; by Omori and Maruyama in OperationRecorder and OperationReplayer [20]; and by Ebraert *et al.* in ChEOPS [9].

We extended the later approach with our tool prototype ChEOPSJ[1], which is a Java version of the same model [34, 33]. It operates in the Eclipse background and silently records the

[1]ChEOPSJ: Change and Evolution Oriented Programming Support for Java (`http://win.ua.ac.be/~qsoeten/other/cheopsj/`)

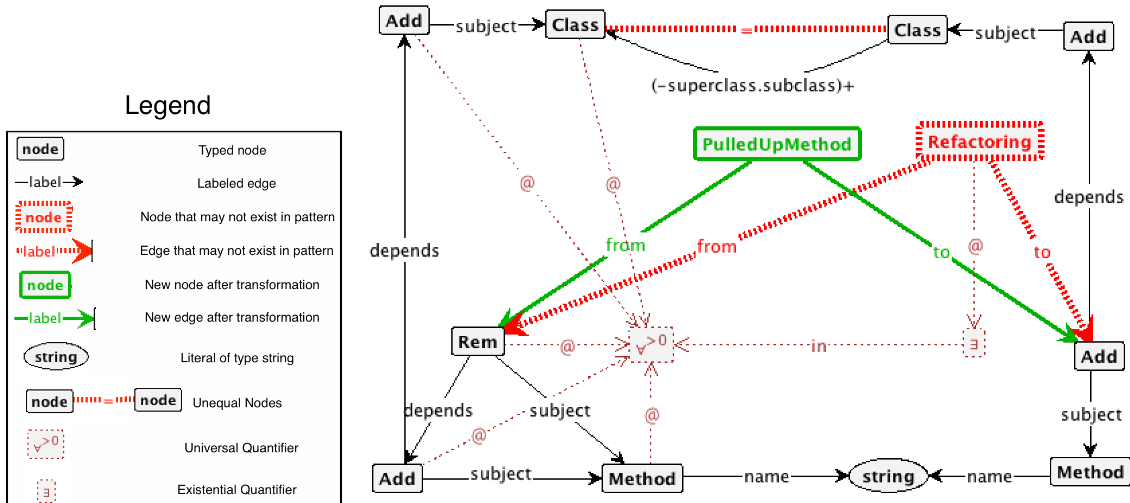


Figure 3: The graph transformation rule to reconstruct the PullUpMethod refactoring

changes that are made to the source code while the developer is programming.

In our tool we implemented two kinds of *Atomic Changes*: `Add` and `Remove` (see Figure 1). These act upon a *Subject* that represents an actual entity in the source code. For these subjects we implemented a subset of the FAMIX model [5] (see Figure 2). We chose the FAMIX model as this model captures most object oriented programming languages. It defines entities representing packages, classes, methods and attributes, as well as more fine grained entities such as method invocations, variable accesses and inheritance relationships.

In our change model the changes are interconnected through dependencies. These dependencies between change objects (See Figure 1) are determined by the relationships between the entities in the subset of the FAMIX model we are using (See Figure 2). Hence, the dependencies between change objects are defined as follows: A change c_1 is said to depend on another change c_2 if the application of c_1 without c_2 would violate the system invariants. For instance, an addition of a method depends on the addition of a class as you cannot add a method to a nonexistent class. As such a software system and its entire evolution is represented as a graph with the changes as nodes and the dependencies between the changes as edges.

Once the sequence of changes and their dependencies is recorded, we use Groove [24], a graph transformation tool, to search the change graph for pre-defined patterns corresponding to a refactoring. We chose Groove because it uses a simple XML format to store their graphs, as such it was easy to export the change graphs from ChEOPSJ into a Groove readable format. Besides that, Groove offers a fast and scalable state space exploration so it should be able to find our refactoring patterns on large graphs relatively quickly.

As an example, we briefly describe how we reconstruct a PullUpMethod refactoring from a graph of changes. The other refactorings are defined in a similar way and are published on **figshare** [29]. The Groove graph transformation is shown in Figure 3. The top of this pattern describes how the classes are related. The class from which the method is removed needs to be different from the class in which we added a method. The class node on the left should be a descendant to the class on the right. We express this relationship using a regular expression `(-superclass.subclass)+`, meaning that we traverse the edge in the opposite direction (with `-superclass`) from the superclass node to the implicit Inheritance node, and then traverse the edge in the normal direction (`subclass`) to the subclass node. Adding the `+` makes this the transitive closure, meaning that we can trace this edge to any descendant class in the inheritance tree of the superclass. The bottom half of the pattern describes the changes to the methods. In the subclass the method has (at least) two changes: an addition (which is dependent on the addition of the class) and a removal which is dependent on the addition of the method. In the superclass the method has (at least) one change: an addition. Moreover the method in the subclass and the superclass should have an identical name. The left side of the pattern (the subclass and its method along with the additions of both and the removal of the method) have a universal quantifier (∀) meaning that this pattern applies to all subgraphs of this kind. In other words, for each instance of a removal of the method in a subclass, this removal is part of the PullUpMethod refactoring reconstructed. This graph transformation rule adds a new node – PulledUpMethod – linked to (i) the changes that remove instances of the method in subclasses (ii) the change that adds the method to the superclass. However this should only be done if these changes are not already linked to a previously reconstructed refactoring node.

We argued the feasibility of a change-based approach for refactoring reconstruction in a previous paper where we had implemented a way for detecting MoveMethod and RenameMethod [35]. For this paper, we extended this proof by construction to incorporate 11 of the refactoring rules that can be expressed on the model in Figure 1 and Figure 2.

- PullUpMethod
- PullUpField
- PushDownMethod
- PushDownField
- MoveClass
- MoveMethod
- MoveField
- RenamePackage
- RenameClass
- RenameMethod
- RenameField

With this list, we have a sufficient basis to compare against the state-of-the-art snapshot-based tool RefFinder [14].

3. REFACTORING MASKING

In this section we address **RQ 1:** *Under which conditions does a snapshot-based approach fail to reconstruct refactorings?* We illustrate the refactoring masking phenomenon, by using the state of the art tool RefFinder [14] on a small yet realistic system: the LAN Simulation [6][2]. This is a script of refactorings that are performed on a small system. It is mostly used as a teaching lab to teach how and why to refactor.

3.1 Experimental Setup

We followed the script of the LAN Simulation [6] and injected some non-refactoring changes along the way. After every atomic change we committed revisions to a local subversion repository. For these commits, we handled the same level of granularity (method level changes) as the model in our change recording tool shown in Figure 2. For instance when performing a MoveMethod refactoring, we executed a simple copy, paste and delete and then updated the signature and the invocations. This resulted in at least 5 commits to the repository: after the copy; after the paste; after the delete; after the signature update; and after the invocation update. As such we created a fine grained change model stored in a subversion repository, with each revision containing one change. We then had RefFinder compare each revision with every other revision in order to find both the smallest and the largest distance between revisions needed to reconstruct a refactoring.

3.2 Results

We performed a series of 22 refactorings in 150 commits: 2 instances of IntroduceExplainingVariable; 7 instances of ExtractMethod; 10 instances of MoveMethod; 2 instances of ExtractSubclass and a single instance of ReplaceConditionalWithPolymorphism). The repository is published on **figshare** [30]. We compared all possible pairs of these 150 commits with RefFinder. That is, we compared revision 1 to revisions 2 to N, then we compared revision 2 to revisions 3 to N, and so on. We looked at the refactorings that RefFinder reconstructed all together, and summed up all the unique distinct refactorings it could reconstruct in all pairs of revisions. We found that RefFinder reconstructed 100 refactorings, of which 40 were false positives, 19 were true positives and 41 were neither, but could be considered as side effects (or subrefactorings) of the performed refactorings. For instance, a MoveMethod usually also involved the removal of a parameter, as the class to which the method was moved used to be a parameter. Another example was the ExtractSubClass refactorings, that also consisted of 3 MoveMethod refactorings. Additionally there were three false negatives: two instances of ExtractMethod and one instance of MoveMethod refactorings that we performed, but that RefFinder did not manage to reconstruct.

The important thing to note is that most of the refactorings we performed were reconstructable by RefFinder at one point or another. In Figure 4 we show the minimum and maximum windows under which these occur. The maximum window is shown below the minimum window; the latter shows the revisions where the refactoring was actually performed. The maximum window denotes up to which point, either before or after the minimum window boundaries, RefFinder is capable of identifying the performed refactoring. To the right of the figure, each refactoring is numbered for easy referencing in the following paragraphs.

We see that refactorings 7, 8, 13, 16, 17, 18, 19 and 20 have a window that reaches to the HEAD revision, which implies that these refactorings were not masked by any other changes. The other eleven refactorings are at one point masked by other changes, hence no longer reconstructable.

The first refactoring masking instances that we want to highlight are the refactorings 3, 6, 10, 12 and 14. These are one ExtractMethod and four MoveMethod refactorings that are masked by changes other than refactoring operations. The ExtractMethod was no longer reconstructable, since we completely changed the code inside the extracted method. It was changed in such a way that it semantically did more or less the same thing, but syntactically it was completely different. A similar observation can be made with the four MoveMethod refactorings that are masked by non refactoring changes.

The other six refactoring instances are masked by other refactoring operations (indicated with the dashed lines). The first are the two IntroduceExplainingVariable refactorings (refactorings 1 and 2), these were hidden by MoveMethod 3 (refactoring 8), as this move operation moved the method in which the variables were introduced. The figure then also shows that four ExtractMethod refactorings were hidden by a MoveMethod refactoring. What happened is some code was extracted to a method and this newly created method was then moved to a different class, at which point it was no longer possible to reconstruct the `ExtractMethod` refactorings. A special case is Extract Method 2, which is shown twice in the figure (as refactorings 4 and 5). This is because this refactoring extracted a duplicate series of statements from two methods into a new method. RefFinder identified this as two distinct ExtractMethod refactorings.

We conclude from this experiment that the minimum window in which a refactoring can be identified needs to be comprised of, at least, the changes of that refactoring. The maximum window in which the refactoring can be reconstructed is uncertain, as a refactoring can be hidden by other operations. In our case study, we have observed that a refactoring can always be reconstructed as long as no other changes act on the same source code entities as that refactoring. One could argue that it is possible to write new rules for RefFinder that identifies a combination of refactorings, but this is an infeasible approach since we can not be expected to devise rules for every possible combination of refactoring operations.

*As an answer to **RQ 1**, we conclude that a snapshot-based approach fails to reconstruct refactorings when other edit operations act on the same source code entities. We then say that the refactoring is* “masked” *by those other changes.*

4. COMPARING REFFINDER TO CHEOPSJ

In this section we address **RQ 2:** *Do fine-grained changes allow us to reconstruct refactorings where snapshot-based approaches fail?* We search for instances of refactoring masking in two real-world cases and see whether a change-based approach is capable of reconstructing refactorings where snapshot-based approaches do not.

[2] `http://lore.ua.ac.be/Research/Artefacts/refacLab/`

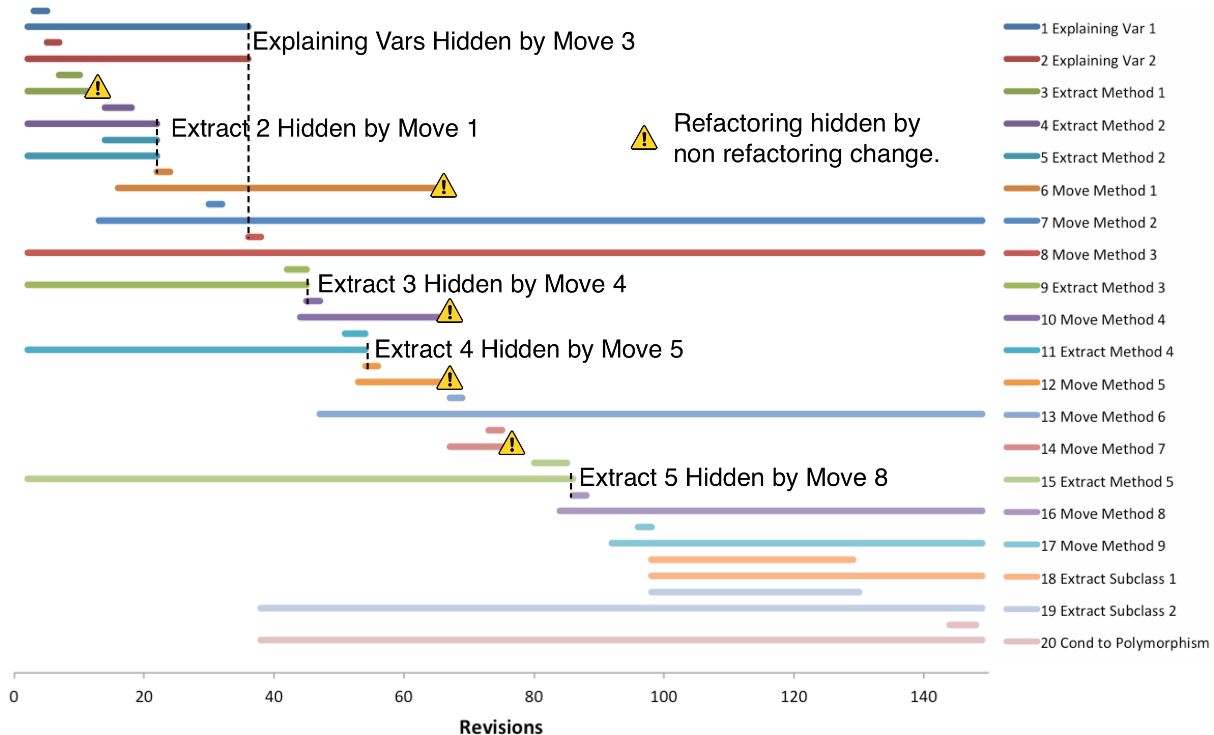


Figure 4: Minimum and maximum windows in which the performed refactorings are reconstructable by RefFinder.

4.1 Experimental Setup

We used two larger open source cases—PMD and Cruisecontrol. PMD[3] is a source code analyser to find a variety of common mistakes like unused variables, empty catch blocks, unnecessary object creations, and so on. Cruisecontrol[4] is a framework that allows for creating a custom continuous integration process.

These projects were selected as they are written in Java and their development history is freely available on a subversion repository. More importantly, the developers of these two software projects used the commit messages to consciously document some of the applied refactorings. We could thus mine these commit messages using a simple grep command to identify those revisions with documented refactorings. We searched for commit messages that contain the terms “refactor(-ed,-ing)”, “move(-d)” or “rename(-d)”.

We sampled these two projects for cases of refactoring masking and selected 10 revisions containing masked refactorings. Tables 1 and 2 show the selected revisions in which we have a total of 26 masked refactorings. These revisions were selected with the following criteria: each revision should be one with refactoring documented in the commit message; RefFinder should be unable to reconstruct any refactorings; and a manual analysis of the revision should show that there were actual refactorings performed and that RefFinder’s refactoring reconstruction failed due to refactoring masking. Note that this is a very conservative way of looking for cases of refactoring masking, since we cannot claim that there is no refactoring masking in either the revisions that had no documented refactorings or the revisions where RefFinder did find refactorings.

[3]`http://pmd.sourceforge.net`

[4]`http://cruisecontrol.sourceforge.net`

In these cases of refactoring masking, we set out to reperform those refactorings while recording our changes with ChEOPSJ in order to obtain a fine grained change model. Suppose revision R_x is documented as a refactored version of revision R_{x-1} and RefFinder cannot identify the performed refactorings because they were masked by other changes. In this case we checked out revision R_{x-1} and distilled an addition change for every source code entity in this revision to have a starting set of changes. We then started recording the changes and performed the refactorings that we identified during our manual analysis in order to obtain the fine-grained change history. At the end of re-performing the refactorings, we verified (using Eclipse’s built in compare support) that our changed version of the system matched revision R_x. We could then export our graph of recorded changes to a Groove readable format and had Groove perform our graph transformation rules to reconstruct the refactoring instances.

4.2 Refactoring Masking Instances

In the PMD project we analysed 6 revisions, where RefFinder was unable to reconstruct the refactorings performed due to refactoring masking. The details of these six revisions are in Table 1. In all of the cases, the masking involved an ExtractMethod refactoring followed directly by a MoveMethod or PullUpMethod refactoring. This means that the developers extracted a piece of code and immediately moved it to a class where they felt it belonged. RefFinder is unable to reconstruct either of these refactorings. For the ExtractMethod it looks for a newly created method in the same class, but the method is already moved to a different class. For the MoveMethod it looks for the class from which the method originates, but there was no such method to begin with.

Table 1: Refactoring masking in PMD.

Rev.	Commit Message	Refactorings Performed
578	**refactor**ing	ExtractMethod MoveMethod
659	more CPD-aided **refactor**ing	ExtractMethod PullUpMethod 2xInlineTemp
1082	more **refactor**ing and tests	ExtractMethod MoveMethod InlineTemp
1085	more **refactor**ing	ExtractMethod MoveMethod
1088	minor **refactor**ing	ExtractMethod MoveMethod RenameLocalVar
2207	**Refactor**ed some more code into `CommandLineOptions`, wrote some more tests	ExtractMethod MoveMethod
	Number of refactorings masked	16

A typical example is shown in listings 1 and 2. An ExtractMethod extracted some code from the `visit` method in the class `EmptyFinallyBlockRule` into a new method called `getFinallyBlock`. This method was then moved to the class `ASTTryStatement`.

```
package net.sourceforge.pmd.rules;
public class EmptyFinallyBlockRule extends AbstrRule {
  public Object visit(...){
    //some Code that is extracted and moved
  }
}

package net.sourceforge.pmd.ast;
public class ASTTryStatement extends SimpleNode {
}
```

Listing (1) PMD revision 1084.

```
package net.sourceforge.pmd.rules;
public class EmptyFinallyBlockRule extends AbstrRule {
  public Object visit(...){
    ASTBlock finallyBlock = node.getFinallyBlock();
  }
}

package net.sourceforge.pmd.ast;
public class ASTTryStatement extends SimpleNode {
  public ASTBlock getFinallyBlock() {
    //some Code that is extracted and moved
  }
}
```

Listing (2) PMD revision 1085.

In the case of Cruisecontrol we analysed 4 revisions containing instances of refactoring masking. We detailed these revisions in Table 2. One of these (rev 842) is similar to the refactoring masking instances in PMD, in that a new method was extracted and immediately pushed to the responsible subclass. The one difference is that the extracted method had the same identifier as the method from which it was extracted and the class to which it was moved was a subclass. As such they created a polymorphic version of the same original method. To maintain behaviour and thus make it a pure refactoring, they added a call to the super

Table 2: Refactoring masking in Cruisecontrol.

Rev.	Commit Message	Refactorings Performed
842	Making `buildResultsUrl` optional in `HTMlEmailPublisher`. **Move**d validation of the parameter from `EmailPublisher` to `LinkEmailPublisher`.	Combination of ExtractMethod PushDownMethod RenameMethod
879	**rename**d `_now` and `getNow()` to `timeOfCheck` and `getTimeOfCheck()`	RenameMethod RenameField
2257	**move** knowledge of default log location into `Log` class	Combination of ExtractMethod MoveMethod InlineMethod
2629	... Also **rename**d `IO.deleteFile` to `IO.delete` because the File part is obvious from the signature and therefor superfluous....	RenameMethod RenameParameter
	Number of refactorings masked	10

implementation. Another similar operation was in revision 2257, where they moved a few statements from one method to another method in a different class. One way of doing this is simply cutting and pasting the code from the one method to the other, which is probably what the developers did. However this same result could also be achieved by performing an ExtractMethod and a MoveMethod to get the piece of code to the right class. Then the invocation to the new method needs to be removed from the original method and an invocation needs to be added in the place where we want the code. To finish off an InlineMethod would put the code where we want it.

Cruisecontrol also provided us with a few examples of masked RenameMethod refactorings. In one case (revision 879), this refactoring is hidden by a RenameField refactoring. In RefFinder the rule to reconstruct a RenameMethod checks whether the method body has a certain similarity. In this case the method body consisted of a single statement: a return statement returning the value of the field. Since the field itself was also renamed to a name that is very different, the method body was no longer similar enough to count as a RenameMethod.

4.3 Change-Based Reconstruction

Our approach for refactoring reconstruction based on recorded changes was capable of reconstructing all refactorings for which we currently have rules. More precisely, our approach allowed us to identify 12 out of 26 refactorings that, for RefFinder, were masked.

Specifically we could reconstruct all of the MoveMethod refactorings and the one PullUpMethod refactoring that we performed in PMD as well as the two MoveMethod, the two RenameMethod and the one RenameField refactorings we performed in Cruisecontrol. As an example we present the results of the refactoring pattern reconstructed from the changes we performed to go from revision 658 to revision 659 in PMD (Figure 6) and the patterns reconstructed from the changes between revision 878 and 879 in Cruisecontrol (Figure 7). All other resulting graphs can be

found on figshare [31]. Figure 6 shows that there was a PullUpMethod refactoring that removed two instances of a method named "`getEndName`" in two subclasses of the class "`UnusedCodeRule`" and an addition of a method by the same name in the superclass. Figure 7 shows that in Cruisecontrol we were able to reconstruct two refactorings from a set of changes: one is the RenameField refactoring that removes the attribute `_now` and adds the attribute `timeOfCheck`; and the RenameMethod refactoring that changes the name of this attribute's getter from `getNow` to `getTimeOfCheck`.

*We conclude that the presence of fine-grained changes allows us to reconstruct refactorings where snapshot-based approaches fail. Indeed, we have found several occurrences of masked refactorings in two real-world open sourced cases, all of which RefFinder was unable to reconstruct. In contrast, our change-based approach was able to reconstruct 12 out of 26 masked refactorings; the other refactorings could be reconstructed by extending the source code model (see Figure 2) and defining new rules for these particular refactorings. As such we effectively answered **RQ 2**.*

4.4 Is this relevant?

Knowing that change-based approaches are capable of reconstructing refactorings where snapshot-based approaches fail, the natural follow up question is to what extent is this improvement relevant. That is, how often does refactoring masking occur in real software projects? This question is impossible to answer precisely, given that there is no project where all refactoring operations are recorded [17]. Moreoever, Negara *et al.* already reported that on average 30% of refactoring operations do not reach the version control system [18]. Nevertheless, we can make a rough estimate based on the data gathered by the Eclipse Usage Data Collector (UDC)[5]. This data is made publicly available and Emerson Murphy Hill et. al. have put the whole data set on Google BigQuery, which enables us to query and process this data using Google's storage and compute infrastructure [28].

We know that snapshot-based approaches typically fail when several edit operations act on the same source code entities. In particular, sequences of ExtractMethod, MoveMethod (and in the case of Cruisecontrol also a RenameMethod) operating on the same segments of code are likely to cause misses. From the UDC dataset it is clear that the RenameMethod operation is by far the most used automated refactoring in Eclipse; Move–Element– and ExtractMethod are also among the top most used automated refactoring operations. This gives a first indication that refactoring masking is relevant.

We are most interested in the combination of ExtractMethod and MoveMethod, so we looked at the edit operations that occurred within 5 minutes after an ExtractMethod operation (see Figure 8). Here we assume programming locality, that is two edit operations that occur closely together are likely to act on the same source code entities [23]. Move–Element– appears at the nineteenth position, but the top three operations are Delete, Paste and Copy; which serve as a manual substitute for a move. Launching a query which counts all ExtractMethod operations that are followed within 5 minutes by either a MoveMethod or by a Copy, a Paste and a Delete, we found 10,869 instances out of a total of 43,602, thus almost 25%. Note that this is a conservative estimate, as these are the situations where we know a snapshot-based approach will fail. In reality, there are likely to be more, since we only took into account those change sequences which start with an extract method. Therefore, we argue that a snapshot-based approach is likely to miss a significant amount of the refactoring sequences, hence that the potential improvements induced by a change-based approach are indeed relevant.

[5]http://www.eclipse.org/epp/usagedata

5. THREATS TO VALIDITY

We now identify factors that may jeopardise the validity of our results and the actions we took to alleviate the risk. Consistent with the guidelines for case studies research [26, 40] we organise the identified threats into four categories.

Construct validity – do we measure what was intended. We relied on the versioning system's log messages to identify revisions corresponding to refactorings. Since no strict conventions are in place for what should be specified in such messages, there may be significant differences in the content and quality of log messages across tasks and developers. Consequently, we might miss certain revisions which do correspond to refactorings. However it was never our intent to find all instances of refactorings that occurred in the system's evolution. In that sense, using this simple way of locating instances of refactorings is sufficient for our purposes.

An additional threat could be that the expertise and experience of developers plays a key role in this application. A developer that knows the purpose of the different refactorings, might be less inclined to perform floss refactoring and actually commit the refactoring as a whole to the source code management system.

Moreover, the fine-grained recorded changes did not exist in the original sample from the repositories so we had to manually re-perform them. These changes might not be the ones applied by the developers. The transformations we performed, in order to obtain the fine-grained change history, are just one possible change-sequence from many potential scenarios. We used our expertise to choose the transformations that we would have applied. We verified (using Eclipse's built in compare support) that our changed version of the system matched the actual revision in the repository.

Internal validity – are there unknown factors which might affect the outcome of the experiment. The substitution of "real" recorded changes by a manual synthetic reproduction of them might be a confounding factor for our results. The improvement in the number of masked refactorings detected by our approach over RefFinder might not be due to the availability of fine-grained changes but to the particular change sequences we manually applied. To reduce this risk, the first two authors of this paper proposed and discussed these change sequences in order to come up with the transformations that, in our opinion, are closest to the real ones.

As already mentioned, there can also be a problem of selection bias caused by how we decided on the refactored revisions we used as experiment subjects. The need to verify

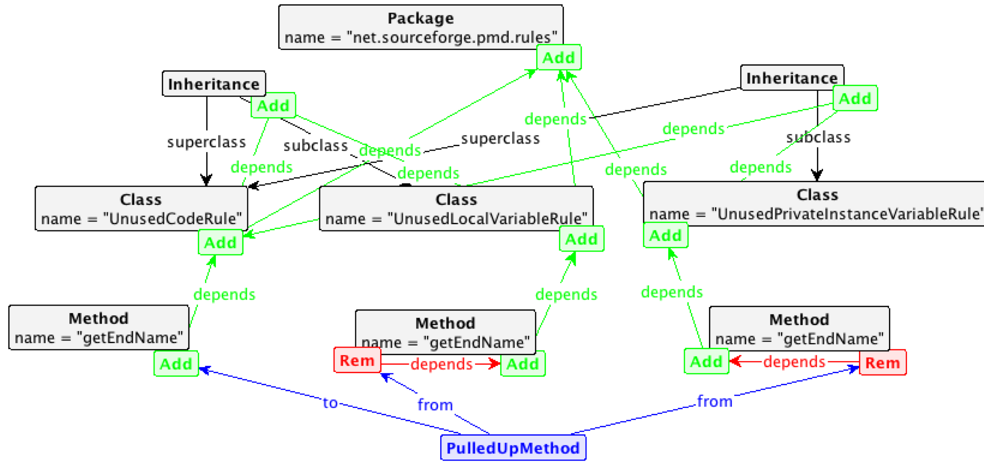


Figure 6: The PullUpMethod refactoring identified by ChEOPSJ in PMD. (note: The subject relationships between changes and source code entities are hidden for the sake of readability.)

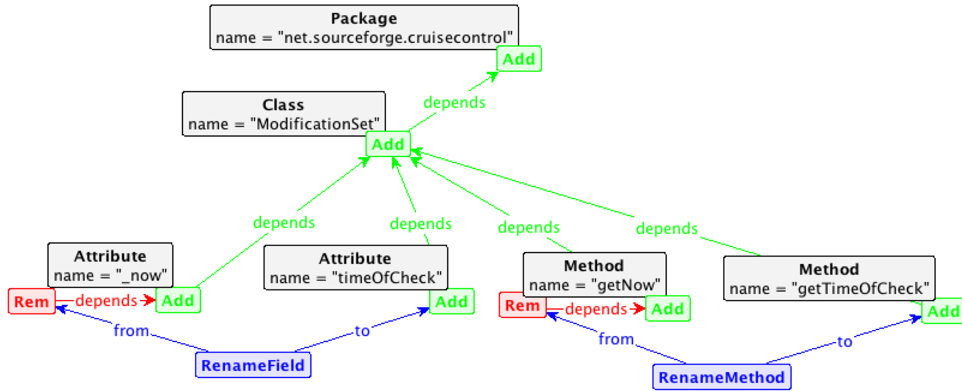


Figure 7: The RenameMethod and RenameField refactorings identified by ChEOPSJ in CruiseControl. (note: The subject relationships between changes and source code entities are hidden for the sake of readability.)

RefFinder results and to manually apply the fine-grained changes, led us to sample only those revisions whose commit messages mentioned refactoring operations. In order to alleviate the potential bias, we run the experiments in a toy example and in two different open source systems. Similar results were obtained from them.

External validity – to what extent is it possible to generalise the findings. In this study we investigated two cases: Cruisecontrol and PMD. We chose them to be sufficiently different, yet, with only two data points, we cannot claim that our results generalise to other systems. The results are also dependent on the number of refactoring detection rules we have implemented. The instances of refactoring masking we have analysed might very well appear in other systems. We cannot however make any claims about other types of (as yet unidentified) refactoring masking.

According to [19], changes in snapshot-based versions are often obscured by other changes. We can expect the refactoring masking problem to be more prominent than what we have inspected. We should also expect different behaviours when developers commit to SVN or Git. It has been observed that programmers commit more often to Git repositories, resulting also in smaller commits [2]. The need for a change-recording mechanism might not be so important in the context of Git repositories, although it also depends on whether the developers use commit squashing (grouping several related changes in one single commit).

Reliability – is the result dependent on the tools. We used RefFinder to construct the baseline for our experiments. The tool might have produced false positives and false negatives, wrongly identifying some refactoring while missing others. On the one hand, this is the best existing tool for refactoring detection. Therefore, despite of the possible errors, it still serves well as a baseline to compare with. On the other hand, we have manually verified the refactorings detected by RefFinder thus, reducing the risk of false positives.

In order to implement our approach, we relied on tools of our own making as well as some external tools. Our ChEOPSJ tool is implemented as an Eclipse plugin and relies on Eclipse's internal java model which can be considered to be a reliable tool. In order to reduce the bias caused by possible bugs and errors in the tool, we tested it extensively. Members of the research groups and some developers at a partner company installed ChEOPSJ and acted as beta-testers for three months. This resulted in many bug fixes

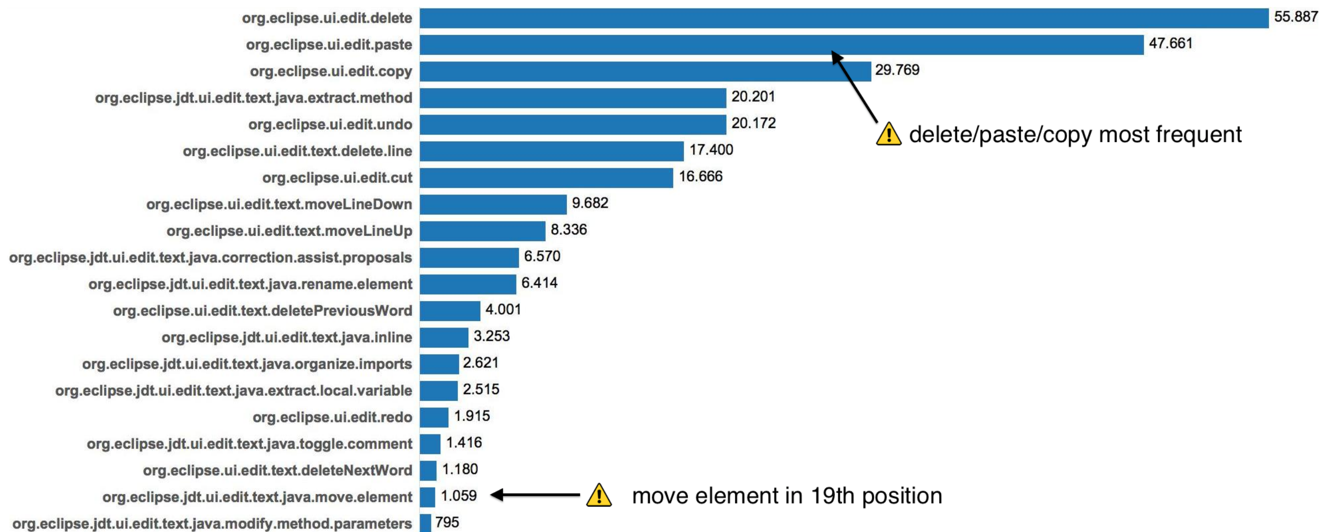


Figure 8: Top 20 most used edit commands within 5 minutes after an ExtractMethod.

and led to a stable and robust version of the tool. For the graph pattern matching we used the tool Groove, which is still actively being developed and improved, but since it has already evolved to a stable and robust tool it too can be considered reliable.

6. CONCLUSIONS

In this paper we have shown how a software evolution history comprised of fine-grained recorded changes can be exploited to reconstruct refactorings more accurately than the state-of-the-art snapshot-based technique RefFinder. To provide a more detailed summary of our findings, we review the research questions we have addressed:

RQ 1 *Under which conditions does a snapshot-based approach fail to reconstruct refactorings?* Snapshot-based approaches fail when other edit operations act on the same source code entities. In particular, combinations of ExtractMethod and MoveMethod confuse a snapshot-based approach, since the effect of the former is concealed by the latter. We then say that the refactoring is *"masked"* by those other changes. Since such simultaneous edit operations might happen at any time, it is impossible to determine the optimal window of changes where snapshot-based reconstruction will still function properly. Hence, the only alternative to faithfully reconstruct refactorings is to have access to the fine-grained changes applied to the code.

RQ 2 *Do fine-grained changes allow us to reconstruct refactorings where snapshot-based approaches fail?* We sampled the version history of two open source projects where the developers made an effort to explicitly document some of the refactorings applied. In particular, we made an opportunistic sample, selecting versions where simultaneous edits on the same entity were performed. Under these conditions, snapshot-based approaches indeed fail to reconstruct the refactorings, while the change-based approach does succeed. Next, we argued that these conditions occur frequently: we estimate that for 25% of all the ExtractMethod refactorings are followed by either a MoveMethod or by a Copy, a Paste and a Delete

Contributions. Over the course of this research, we made the following contributions:

- We implemented a tool prototype named ChEOPSJ serving as an experimental platform for conducting feasibility studies with first-class representation of changes in Java.
- We demonstrated how this platform can be used to reconstruct refactoring operations from a stream of changes.
- We applied the prototype to two cases — PMD and CruiseControl — to compare a change-based approach against a snapshot-based approach.
- We demonstrated that the change-based approach is more accurate than a snapshot-based approach.
- We argued that this improved accuracy is relevant by estimating the number of edit operations acting on the same source code entities within 5 minutes after an ExtractMethod refactoring.

Future work. Our plan for the immediate future is to gather real recorded data and replicate our experiments, thus moving from In-Vitro to In-Vivo research [4]. We are currently deploying the change-recording plugin at some partner companies which should result in detailed streams of changes where we can interview the developers about the details of the refactorings. Next we plan to implement additional detection rules for other refactorings besides the 11 listed under Section 2.2, to investigate other refactoring masking conditions. Particularly, interesting in that respect would be a more detailed change model fully representing AST entities below the method signature level.

*Our findings together with similar results obtained by others (*i.e., *Spyware [25], Syde [12], Cheops [9], OperationRecorder [20]), indicate that it is not only feasible but also worthwhile to maintain fine-grained evolution histories of software projects. Given the popularity of Git, which encourages a fine-grained commit behaviour, having an explicit representation of the changes is the natural successor to the current generation of distributed version control systems.*

7. ACKNOWLEDGMENTS

This work has been sponsored by (i) the Interuniversity Attraction Poles Programme - Belgian State - Belgian Science Policy, project MoVES; (ii) the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen) under project number 120028 entitled "Change-centric Quality Assurance (CHAQ)". We hereby express our gratitude to Arend Rensink for his quick response on the Groove discussion forum[6], as such helping us out with the Groove Syntax.

8. REFERENCES

[1] A. Abran, P. Bourque, R. Dupuis, J. W. Moore, and L. L. Tripp. *Guide to the Software Engineering Body of Knowledge - SWEBOK*. IEEE Press, 2004.

[2] C. Brindescu, M. Codoban, S. Shmarkatiuk, and D. Dig. How do centralized and distributed version control systems impact software changes? In *Proceedings ICSE, 2014*, pages 322–333.

[3] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *Proceedings OOPSLA, 2000*, pages 166–177.

[4] S. Demeyer, A. Lamkanfi, and Q. D. Soetens. "in vivo" research in software evolution. *ERCIM News*, 2012(88), 2012.

[5] S. Demeyer, S. Tichelaar, and P. Steyaert. FAMIX 2.0 - the FAMOOS information exchange model. Technical report, University of Berne, 1999.

[6] S. Demeyer, F. Van Rysselberghe, T. Girba, J. Ratzinger, R. Marinescu, T. Mens, B. Du Bois, D. Janssens, S. Ducasse, M. Lanza, M. Rieger, H. Gall, and M. El-Ramly. The LAN-simulation: a refactoring teaching example. In *Proceedings IWPSE, 2005*, pages 123–131.

[7] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automated detection of refactorings for libraries and frameworks. In *Proceedings WOOR, 2005*

[8] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automated detection of refactorings in evolving components. In *Proceedings ECOOP, 2006*, pages 404–428

[9] P. Ebraert, J. Vallejos, P. Costanza, E. V. Paesschen, and T. D'Hondt. Change-oriented software engineering. In *Proceedings ICDL, 2007*, pages 3–24.

[10] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2001.

[11] C. Görg and P. Weißgerber. Error detection by refactoring reconstruction. In *Proceedings MSR, 2005*

[12] L. Hattori and M. Lanza. Syde: A tool for collaborative software development. In *Proceedings ICSE, 2010*, pages 235–238.

[13] J. Henkel and A. Diwan. CatchUp!: capturing and replaying refactorings to support API evolution. In *Proceedings ICSE, 2005*, pages 274–283.

[14] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit. Ref-finder: a refactoring reconstruction tool based on logic query templates. In *Proceedings FSE, 2010*, pages 371–372.

[15] H. Liu, Y. Gao, and Z. Niu. An initial study on refactoring tactics. In *Proceedings COMPSAC, 2012*, pages 213–218.

[16] A. Murgia, M. Marchesi, G. Concas, R. Tonelli, and S. Counsell. Parameter-based refactoring and the relationship with fan-in/fan-out coupling. In *Proceedings. ICST Workshops, 2011*, pages 430–436.

[17] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. *IEEE TSE*, 38(1):5–18, 2012.

[18] S. Negara, N. Chen, M. Vakilian, R. Johnson, and D. Dig. A comparative study of manual and automated refactorings. In G. Castagna, editor, *ECOOP, 2013*, volume 7920 of *LNCS*, pages 552–576.

[6] `http://sourceforge.net/p/groove/discussion/407076/thread/8eb1f2c4/`

[19] S. Negara, M. Vakilian, N. Chen, R. E. Johnson, and D. Dig. Is it dangerous to use version control histories to study source code evolution? In *Proceedings ECOOP, 2012*, pages 79–103.

[20] T. Omori and K. Maruyama. A change-aware development environment by recording editing operations of source code. In *Proceedings MSR, 2008*, pages 31–34.

[21] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim. Template-based reconstruction of complex refactorings. In *Proceedings ICSM, 2010*, pages 1–10.

[22] N. Rachatasumrit and M. Kim. An empirical investigation into the impact of refactoring on regression testing. In *Proceedings ICSM, 2012*, pages 357–366.

[23] V. Rajlich. Modeling software evolution by evolving interoperation graphs. *Ann. Softw. Eng.*, 9(1-4):235–248, Jan. 2000.

[24] A. Rensink. The groove simulator: A tool for state space generation. In *AGTIVE, 2004*, volume 3062 of *LNCS*, pages 479–485.

[25] R. Robbes and M. Lanza. SpyWare: A change-aware development toolset. In *Proceedings ICSE, 2008*, pages 847–850.

[26] P. Runeson, M. Höst, and M. Alshayeb. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 2009.

[27] C. Schofield, B. Tansey, Z. Xing, and E. Stroulia. Digging the development dust for refactorings. In *Proceedings ICPC, 2006*, pages 23–34.

[28] W. Snipes, E. Murphy-Hill, T. Fritz, M. Vakilian, K. Damevski, A. R. Nair, and D. Shepherd. *Analyzing Software Data*, chapter A Practical Guide to Analyzing IDE Usage Data. Morgan Kaufmann, 2015.

[29] Q. D. Soetens. Groove Refactoring Reconstruction Rules, 06 2014, Retrieved 09:48, Jul 06, 2015 (GMT), **figshare** http://dx.doi.org/10.6084/m9.figshare.1070082

[30] Q. D. Soetens. LAN Simulation Fine Grained Changes Repository, 11 2014, Retrieved 09:49, Jul 06, 2015 (GMT), **figshare** http://dx.doi.org/10.6084/m9.figshare.1237061

[31] Q. D. Soetens. Results for Reconstruction on Cruisecontrol and PMD, 06 2014, Retrieved 09:41, Jul 06, 2015 (GMT), **figshare** http://dx.doi.org/10.6084/m9.figshare.1080418

[32] Q. D. Soetens and S. Demeyer. Studying the effect of refactorings: a complexity metrics perspective. In *Proceedings QUATIC, 2010*, pages 313–318.

[33] Q. D. Soetens and S. Demeyer. ChEOPSJ: Change-based test optimization. In *Proceedings CSMR, 2012* pages 535–538.

[34] Q. D. Soetens, S. Demeyer, and A. Zaidman. Change-based test selection in the presence of developer tests. In *Proceedings CSMR, 2013* pages 101–110.

[35] Q. D. Soetens, J. Pérez, and S. Demeyer. An initial investigation into change-based reconstruction of floss-refactorings. In *Proceedings ICSM, 2013*, pages 384–387.

[36] K. Stroggylos and D. Spinellis. Refactoring–does it improve software quality? In *Proceedings WoSQ, 2007*, page 10.

[37] F. Van Rysselberghe and S. Demeyer. Reconstruction of successful software evolution using clone detection. In *Proceedings IWPSE, 2003*, pages 126–130.

[38] P. Weißgerber and S. Diehl. Are refactorings less error-prone than other changes? In *Proceedings MSR, 2006*, pages 112–118.

[39] Z. Xing and E. Stroulia. Refactoring detection based on UMLDiff change-facts queries. In *Proceedings WCRE, 2006*, pages 263–274.

[40] R. K. Yin and M. Alshayeb. *Case Study Research: Design and Methods, 3 edition*. Sage Publications, 2002.

Hierarchical Categorization of Edit Operations for Separately Committing Large Refactoring Results

Jumpei Matsuda, Shinpei Hayashi, and Motoshi Saeki
Department of Computer Science
Tokyo Institute of Technology
Tokyo 152–8552, Japan
{jmatsu,hayashi,saeki}@se.cs.titech.ac.jp

ABSTRACT

In software configuration management using a version control system, developers have to follow the commit policy of the project. However, preparing changes according to the policy are sometimes cumbersome and time-consuming, in particular when applying large refactoring consisting of multiple primitive refactoring instances. In this paper, we propose a technique for re-organizing changes by recording editing operations of source code. Editing operations including refactoring operations are hierarchically managed based on their types provided by an integrated development environment. Using the obtained hierarchy, developers can easily configure the granularity of changes and obtain the resulting changes based on the configured granularity. We confirmed the feasibility of the technique by applying it to the recorded changes in a large refactoring process.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments—*integrated environments*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*version control*

General Terms

Algorithms, Theory

Keywords

Refactoring, edit history, tangled changes

1. INTRODUCTION

In software development using a version control system (VCS), changes following the *commit policy* of the project are worthful for developers. Developers often understand the code differences performed by other developers, and such understandings help them to understand the meaning of changes and to reuse them [12]. A commit policy defines the discipline of commits by developers. e.g., the upper bound of the size of a commit or the number of meaningful changes included in a commit. For instance, the policy of the Task Level Commit [5] is well known and useful. For improving the understandability and/or reusability of changes, Task Level Commit prohibits committing changes of mixed intentional edits. It requires developers to commit changes, each of which includes only one intention.

However, in actual software development, keeping changes following the commit policy is not a trivial task. Many pieces of research [3, 4, 6, 15–17, 19] reported that *tangled changes*, i.e., changes including multiple intentional edits, often occur when developers prepare their commits. In order to follow the used commit policy, developers are required to pay attention to editing their source code carefully in appropriate manner and order and to committing the performed edits on appropriate moments. Although some VCSs have the features to merge commits or split a commit into multiple ones, these features are not intended to prepare appropriate commits from the edits performed by developers, and they require a redundant understanding of the performed changes, which is time-consuming.

In particular, a complicated editing process often happens when applying a large refactoring. A large refactoring consists of multiple primitive refactoring instances. Developers are needed to apply multiple refactoring operations during a large refactoring process if the whole refactoring is not automated by refactoring tools. In addition, developers often apply *floss refactoring* [18]; refactorings are applied together with non-refactoring changes. In such a case, the resulting sequence of changes are tangled; it includes refactoring operations of different types and non-refactoring edits. Packing these changes as a single commit should be avoided due to the low understandability of the resulting change content.

In this paper, we propose a technique for hierarchically grouping edit operations of source code in order to retrieve sets of operations by specifying the granularity of commits according to the used commit policy. In our technique, each branch of the built hierarchy corresponds to a commit of certain granularity. By collaborating with existing integrated development environment (IDE), some special operations including automated refactorings and non-essential changes [16] can be easily recorded and identified. Our technique automatically builds a hierarchy of such operations and utilizes them for constructing commits of multiple levels of granularity. Developers can obtain the changes according to the specified branch nodes by reordering the performed edit operations. Since our reordering procedure carefully

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.

IWPSE'15, August 30, 2015, Bergamo, Italy
ACM. 978-1-4503-3816-5/15/08...$15.00
http://dx.doi.org/10.1145/2804360.2804363

constructs the changes based on the dependency between edit operations, the consistency of the resulting changes is guaranteed when the procedure succeeds.

We have implemented our technique as a plug-in of Eclipse IDE [1]. We regard the refactoring operations provided by Eclipse as the intentional meanings of changes and their similarity of types as the intentional relationship between changes. By applying our technique to an edit history including both automated and manual edit operations, we show the effectiveness of our approach.

The rest of this paper is organized as follows. In the next section, we clarify our background and motivation using an example. Section 3 describes our approach, and Section 4 illustrates the supporting tool implementing our approach. We discuss our case study in Section 5 to show the effectiveness of our approach. Section 6 introduces related work, and we conclude this paper and state our future work in Section 7.

2. BACKGROUND

In software configuration management (SCM), developers commit their changes. A commit has a message that expresses its meaning or purpose in addition to the change content representing how the change modifies the source files. Developers have to understand the commits of other developers from the message and/or the content of the commits.

One factor why tangled changes occur is the load to prepare commits according to the meaningful intentions separately. The meaning and the size of a commit is determined when a developer performs a commit based on the edits from the most recent revision to the present. One of the ways to construct commits for each intention is to edit source code in the specific order for making changes of each task at once. However, this is an annoying and time-consuming task because at least two editing tasks may relate each other. Although existing VCSs enable developers to perform a commit by selecting some of the edited chunks from the whole edits, it is hard for developers to select only appropriate chunks having the same intentional meaning after performing all of the edits.

Therefore, it is useful for developers to reconfigure their changes after performing them when the resulting changes do not follow the compliant commit policy. We call this process *dividing commits*. An illustrative example of the dividing commits is shown in Figure 1. In this figure, edit operations of three different tasks are performed: change of the naming convention, the aggregation of duplicated code, and a fix according to a change request. The left side of the figure shows the fragments of source code before and after applying the edit operations; we can have the left bottom fragment of code when applying the edit operations shown in the right side to the code fragment at the left top of the figure. First, by applying Rename Method refactorings, the names of methods were changed that the new names have a specific prefix in order to follow the changed naming convention. Next, two `if` statements, which are the duplicates, are aggregated into one using Extract Method refactoring. Finally, a behavioral fix according to the given change request was done. For brevity, its related edits are omitted as "· · ·" as the last edit operation in the figure. These changes happen when we apply floss refactorings [18]; when implementing a fix for a change request, such changes have frequently happened in practice [19].

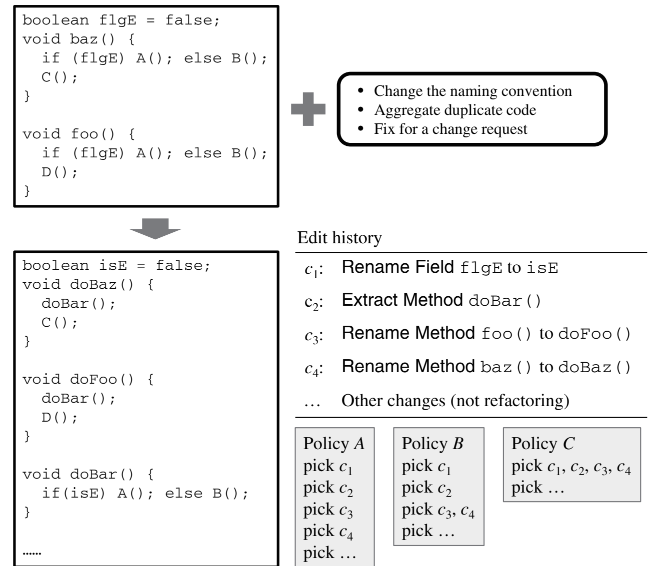


Figure 1: Illustrative example of dividing commits.

How we commit these edits differs according to the commit policy that the target software project follows. For example, consider a commit policy that refactoring operations of different types should be committed separately as different intention of changes in order to describe the commit log as using the type of refactoring as the intention of the change (named Policy B). In contrast, we can also consider another policy that allows us to commit coarser-grained changes for only separating the behavior-preserving changes such as refactorings with the other non-behavior-preserving ones (named Policy C). Moreover, a finer-grained policy compared to Policy B that prohibits to commit multiple refactoring instances at once for improving the understandability of the differences of changes (named Policy A). In Policy A, in addition to the separation based on intentions, developers have to commit each of refactoring instance included in a sequence of intentional changes separately; the policy prohibits to commit so-called *impure* refactoring [9]. This kind of policy leads to increase the number of commits but the ease of the understanding and/or validation of each commit.

Actually, some commit policies define how refactoring-related changes should be divided like the above policies in practice. For example, the commit policy of an open source project named OpenStack [2] suggests to avoid committing mixing functional code changes with whitespace changes: "*The whitespace changes will obscure the important functional changes, making it harder for a reviewer to correctly determine whether the change is correct*". It also suggests to separate refactoring and non-refactoring changes: "*It is highly desirable that any refactoring is done in commits which are separate from those implementing the new feature. This helps reviewers and test suites validate that the refactoring has no unintentional functional changes*". The separation of refactoring and non-refactoring changes improves the understandability of changes when reviewing the resulting commits [8, 13, 23].

Consider that we have a sequence of edits as shown in the table in Figure 1. The lines shown in the right bottom of

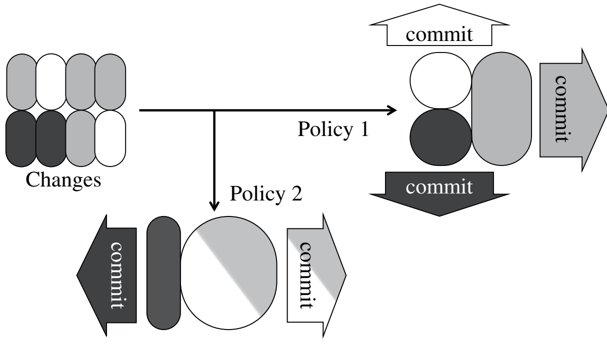


Figure 2: Basic idea.

the figure indicate how the commits in each policy should include which edit operations. For example, the edit operations $\{c_1, c_3, c_4\}$ (a fix for following the new naming convention by Rename Method refactoring), $\{c_2\}$ (the aggregation of code by Extract Method refactoring), and $\{\cdots\}$ (the other changes) should be committed separately in Policy B. This separation does not follow the other policies, i.e., Policies A and C. In Policy C, we have to separate the edit operations like $\{c_1, c_2, c_3, c_4\}$ and $\{\cdots\}$, and in Policy A we have to do like $\{c_1\}$, $\{c_2\}$, $\{c_3\}$, $\{c_4\}$, and $\{\cdots\}$. Committing all the edits shown in the figure does not follow any of these policies. It means that we have to prepare the changes of different granularity based on the compliant commit policy.

Below, we call the granularity of the changes for following the policy the *granularity of a policy*. For example, Policy B treats the applied refactorings as non-behavior-preserving changes and does not split them into multiple refactoring commits. In contrast, Policy A distinguishes the applied refactorings that have the same type from the refactorings of the other types. In other words, Policy A can treat more concrete intentions of changes. In addition, Policy A is finer-grained than Policy B in terms of the granularity of the following changes; the changes following Policy A are finer-grained than those following Policy B.

3. PROPOSED TECHNIQUE

3.1 Our Approach

We propose a technique for reorganizing changes according to different commit policies. Our technique records changes on a structure and enables users to organize changes in different kinds of granularities in order to make the resulting commits follow the granularity of the used commit policy. This makes developers easy to adjust the granularity with keeping the consistency of changes. Since a refactoring-related editing process is the target of this paper, we use the type of refactoring operations for representing the meaning of recorded edit operations.

Figure 2 shows the basic idea of our technique. We record edit operations of source code as finest-grained changes using IDE and reorganize them into meaningful changes according to the used commit policy. If the development project follows a commit policy preferring finer-grained changes, the recorded edit operations are reorganized to finer-grained changes. In contrast, if the development project follows a commit policy preferring coarser-grained changes, the recorded edit operations are merged and reorganized to coarse-grained changes. On one hand in Figure 2, eight finest-grained changes are recorded, and they are categorized into three groups, specified by the colors. In Policy 1, these changes are merged according to their groups, and three commits are organized. On the other hand in Policy 2, since white-colored and gray-colored groups are regarded to the same editing intention, all the changes of these two groups are merged, and coarser two commits are organized. The point is that commits are organized after all the edit operations are performed, which mitigates the cost of developers for paying attention to the manner and order of their code editing.

The overview of our technique is illustrated in Figure 3. We regard an editing operation such as lexical addition or deletion to source code by a developer as a fine-grained change to be recorded. The recorded changes are reordered according to their intentions. For the separation of changes by intentions, we introduce a hierarchical grouping method, which handles a group as a node and a meaningful parent-child relationship as an edge between nodes. The node closer to the root node has the meaning of more abstract than those far from the root node. We convert a commit policy to a set of nodes in the hierarchy representing how we construct commits. After selecting the nodes based on the commit policy, we extract the ordering relation between nodes and reorder the changes belonging to the nodes so that we realize the commits appropriate for the given policy. Finally, the obtained changes are committed to underlying SCM repository such as Git.

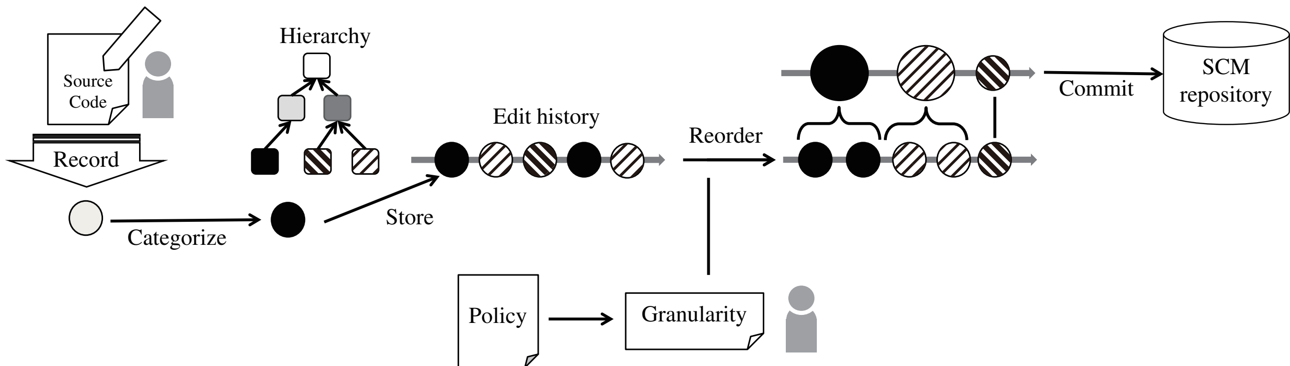


Figure 3: Overview of the technique.

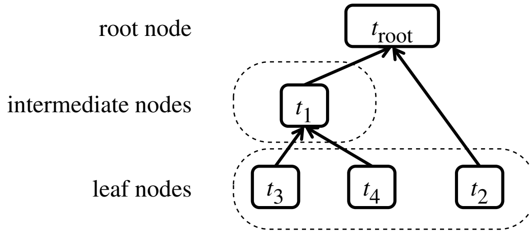


Figure 4: Example of the tree structure.

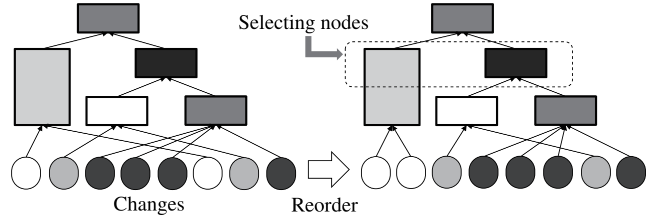


Figure 5: Reorganizing changes.

3.2 Definitions

In our technique, the edit operations of source code are recorded by OperationRecorder [21] and managed by Historef [11]. The definitions among them are based on Historef methodology. We extend the original definition in order for dealing with a hierarchal structure instead of sequential one that the original methodology intended to use.

We call a series of addition, removal, or replace of a specific source code file *chunk*. A chunk $h := (t, f, o, r, a)$ is a tuple representing a modification of the characters on a file of source code, where:

- t is the time when the edit operation was performed,
- f is the file to be edited,
- o is an integer representing the starting offset of the edit,
- r is the removed string from the edited file, and
- a is the added string to the edited file.

Each element of an edit history $H := c_1c_2 \dots c_N$ is a change. A change is a pair of a sequence of chunks $h_1h_2 \dots h_n$ and a group $g \in G$ to which the change belongs: $c = (h_1h_2 \dots h_n, g)$. Each chunk and the size of a change can be respectively referred as $c[i] := h_i$ and $|c| := n$. Some of the dependencies between changes exists. For example, consider an edit history $H = c_{\mathrm{add}}c_{\mathrm{remove}}$ where c_{add} is a change of the addition of string "abc", and c_{remove} is another change of the removal of the string "abc" added by c_{add}. Here, it is impossible to swap these changes because the latter change deletes the text that is introduced by the former one, and we call this situation that the latter change depends on the former change.

We introduce a hierarchical grouping that extends the existing one. The hierarchical grouping consists of a tree structure. We define the node of the hierarchy $t \in T$ corresponding to the group g as $t = (id, caption, t_{parent})$ and redefine a change c as $c := (h_1h_2 \cdots h_n, t)$, where:

- id is the identifier of the node,
- $caption$ is the caption string representing the name of the node, and
- t_{parent} is the parent of the node.

The root node does not have a parent node t_{parent} for avoiding cycling.

As mentioned above, the set of nodes in the hierarchical groups T forms a tree structure. Figure 4 shows an example of such tree structure of the hierarchical groups. This tree structure consists of five nodes: t_{root}, t_1, t_2, t_3, and t_4. The node $t_{root} = (root, caption, t_{root})$ is the *root node* of the tree structure. $T_{leaves} \subset T$ denotes the *leaf nodes* who do not have any child nodes. A node $t \in T$ is called *intermediate* node if it is neither root nor leaf node: $t \notin T_{leaves} \cup \{t_{root}\}$. For example in Figure 4, t_{root} is the root node, t_1 is the intermediate node, and t_2, t_3, and t_4 are the leaf nodes.

All the changes belong to leaf nodes of the tree structure. When a change c belongs to a leaf node t, the relation between the change and the node can be referred by $\mathrm{t}(c) := t$. Although no changes directly belong to intermediate nodes, they indirectly belong to them based on the parent-child relationship between nodes. Therefore, intermediate nodes can be regarded as a kind of abstract categories. All of the changes C_t that belong to a node t and its descendant nodes can be referred by $\mathrm{changes}(t) := C_t$. For example, an instance of Rename Field refactoring to the field f using a feature of an IDE belongs to the node $t_f^{\text{Rename Field}}$, and the actual change representing the replacement of each occurrence of the corresponding identifier can be retrieved by $\mathrm{changes}(t_f^{\text{Rename Field}})$.

Our interest is how to divide the set of all the changes in the leaf nodes into smaller sets of changes. We denote a *division* of T_{leaves} by $T_{sep} \subseteq 2^{T_{leaves}}$, with satisfying the following condition:

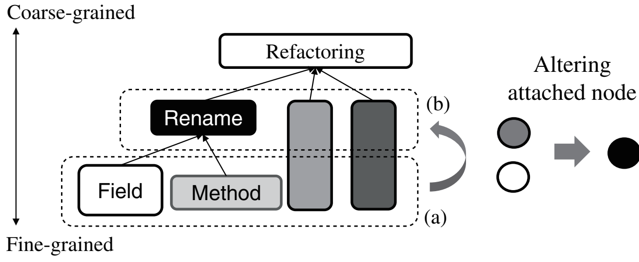
$$\forall t_1 \neq t_2 \subseteq T_{sep} \bullet t_1 \cap t_2 = \emptyset,$$
$$T_{leaves} = \bigcup_{t \in T_{sep}} t.$$

This condition guarantees that T_{sep} covers every element of T_{leaves} and its two optional elements are disjoint.

Since all the changes belong to only the leaf nodes, the order on the leaf nodes is important when reordering changes. A binary relation $\prec \subseteq T_{leaves} \times T_{leaves}$ expresses the total order for reordering the changes on the proposed tree structure. We can prepare this order relation $\prec$ based on the given separation T_{sep}. For example, if two nodes t_1 and t_2 are classified to the same category, i.e., $\{\dots, t_1, t_2, \dots\} \in T_{sep}$, occurring of a node t in another category in T_{leaves} in between these nodes in the order is prohibited: $t_1 \prec t \prec t_2$ or $t_2 \prec t \prec t_1$ should not hold.

3.3 Reorganizing Changes

We reorganize the changes by reordering the edit operations based on the given hierarchy in order to construct the commits following the suitable granularity defined by the used commit policy. Figure 5 shows the overview of reorganizing changes. This procedure consists of the following two steps. First, we select nodes of the given hierarchy according to the suitable granularity. Next, we reorder the edit operations in the order based on the selected nodes. These two steps realize to organize the commits following the policy.



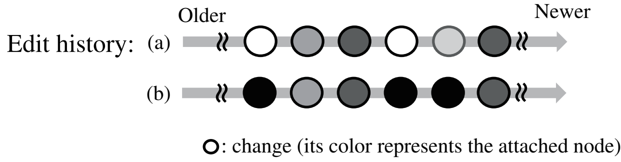


Figure 6: Selecting nodes.

Name: $\mathsf{ReorderH}(H, H', \preceq)$
Input:

$H \equiv c_1 \cdots c_N$: edit history
$H' \equiv c_i \cdots c_j$: target subsequence of H
$\prec \subseteq T_{leaves} \times T_{leaves}$: order relation of leaf nodes

Steps:

1. Sort H' by the bubble sorting based on $\prec$ in order to satisfy the condition $\mathrm{t}(c_i) = \mathrm{t}(c_k) \Rightarrow \mathrm{t}(c_i) = \mathrm{t}(c_j)$ for all $i < j < k$. Swap is used for swapping two changes in the sort.
2. If it is necessary to merge changes, it does. When merging the changes into one, it executes $\mathsf{Merge}(H, c_i \cdots c_j)$ for all $i \cdots j$ s.t. $\mathrm{t}(c_i) = \cdots = \mathrm{t}(c_j)$.

Name: $\mathsf{Swap}(H, c_i, c_j)$
Input:

$H \equiv c_1 \cdots c_N$: edit history
c_i, c_j: Adjacent target changes in H

Steps:

1. $c'_i \leftarrow c_i$, $c'_j \leftarrow c_j$.
2. **for** $k = |c'_i|$ **downto** 1 **do**;
 for $l = 1$ **to** $|c'_j|$ **do**;
 Commute $c'_i[k]$ and $c'_j[l]$ [12].
3. $H \leftarrow c_1 \cdots c_{i-1} c'_j c'_i c_{j+1} \cdots c_N$.

Name: $\mathsf{Merge}(H, c_i c_{i+1} \cdots c_j)$
Input:

$H \equiv c_1 \cdots c_N$: edit history
$c_i c_{i+1} \cdots c_j$: target subsequence of H

Steps:

1. $c' \leftarrow (c_i[1] \cdots c_i[|c_i|] c_{i+1}[1] \cdots c_{i+1}[|c_{i+1}|] \cdots c_j[1] \cdots c_j[|c_j|], \mathrm{t}(c_i))$.
2. $H \leftarrow c_1 \cdots c_{i-1} c' c_{j+1} \cdots c_N$.

Figure 7: Procedures ReorderH, Swap, and Merge.

3.3.1 Selecting Nodes

Selecting the granularity of commits for reorganizing is required when reorganizing commits according to the used commit policy. In our technique, this step is equivalent to selecting nodes. Figure 6 illustrates the change of target granularity by selecting nodes. A circle and a horizontal arrow in the figure respectively represent a change and an edit history. In the edit history, newer changes are located in the right-side. The figure includes an excerpt history consisting of six changes: $c_1, \ldots, c_6$. In Figure 6(a), the user had selected four nodes, t_2, t_3, t_4, and t_5, i.e., four groups of edit operations, and we classify the changes by the selected four types. Then, the user discontinued to treat the rename of methods and fields as different changes and regarded them as an atomic change of abstract and coarser-grained renaming. The nodes of Rename Method and Rename Filed belong to the node of abstract Rename. Changes are belonging to the node of Rename Method or Rename Filed. As shown in Figure 6(b), the user selected coarser-grained nodes, t_1, t_2, and t_3, that regard the renaming as a single intention. By changing the nodes, all of the changes were classified into three groups. Actually, three changes that initially classified into two groups, c_1, c_4, and c_5, were regarded as having the same intention and classified into a single group.

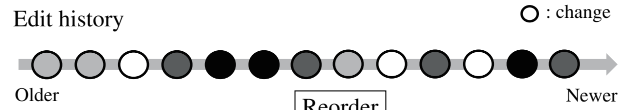


Figure 8: Example of reordering using ReorderH.

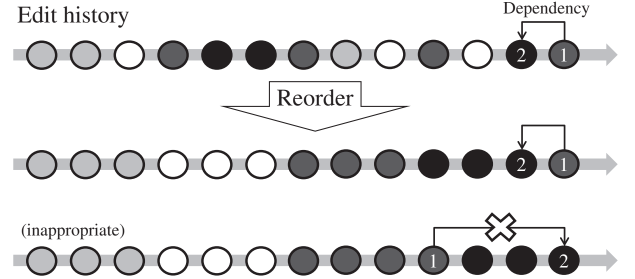


Figure 9: Dependency constraint.

3.3.2 Reordering Edit Operations

We defined another history refactoring operation that reorders the changes on an edit history categorized by the proposed hierarchy. The defined operation ReorderH is an extension of Reorder, the original reordering operation in Historef [11]. We extended Reorder in order to treat hierarchical groups as input and to use the order relation between the leaf nodes of the hierarchy.

The procedure of ReorderH is shown in Figure 7. ReorderH uses two history refactorings defined in Historef: Swap and Merge. Swap swaps two given changes. Merge merges given changes into a single change. By using ReorderH, the given subsequence of the edit history H' is reordered and the changes that belong to the same element of T' are merged into a single change. The resulting merged changes follow the order relation $\prec$. Therefore, the granularity of the resulting changes are defined according to the given order $\prec$. Figure 8 shows the ideal reordering when applying ReorderH to the

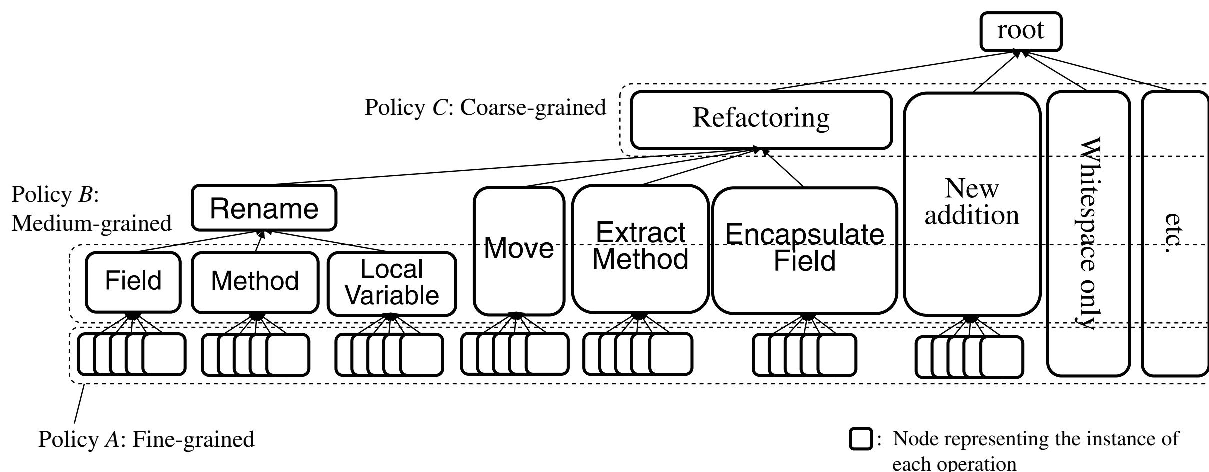


Figure 10: Hierarchical groups.

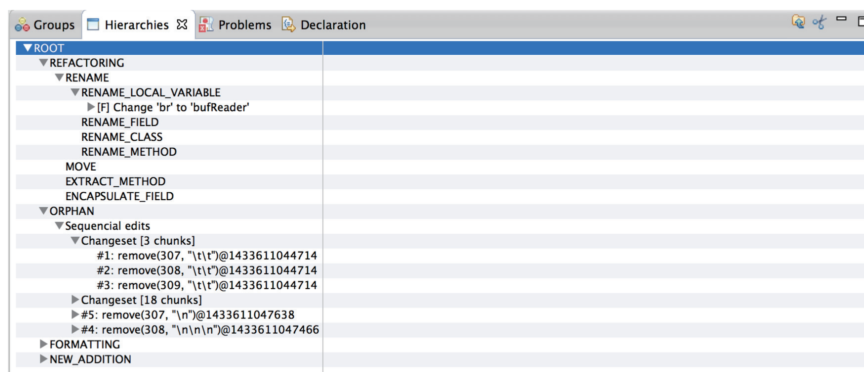


Figure 11: Screenshot of the tool.

edit history H. The arrows of the vertical direction indicate the execution of ReorderH. The changes of the same color will be located to close and adjacent.

Note that there are some dependency relationships between changes. The reordering using ReorderH has to keep the relative order between two changes among a dependency. For example, consider the dependencies shown in Figure 9. As the resulting history of ReorderH, the bottom history is inappropriate because it violates the dependency constraint between the changes 1 and 2, and the middle one is the actual result.

4. IMPLEMENTATION

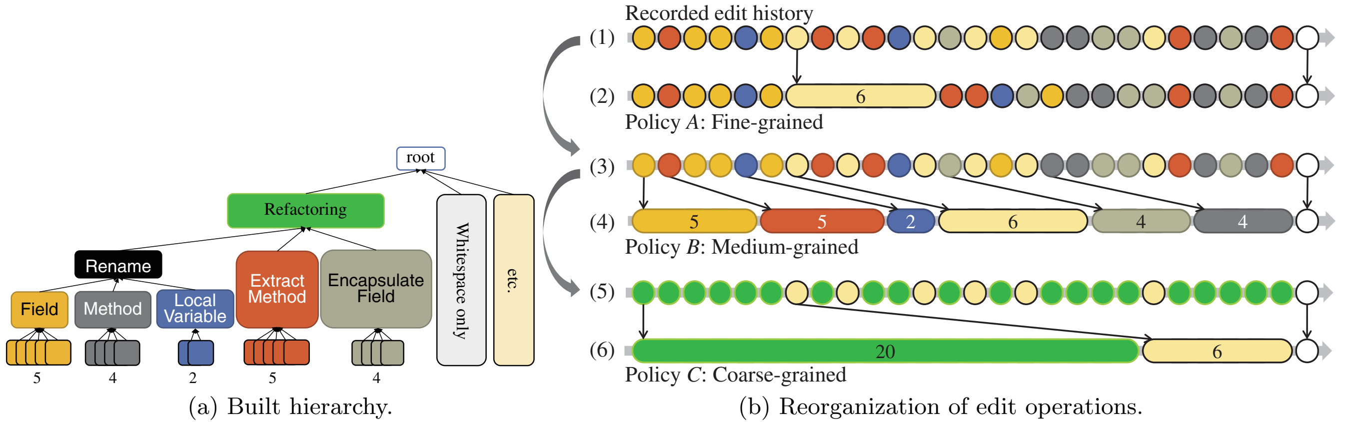
We have implemented a tool for providing an automated support of our approach. The edit operations used in our technique are collected using OperationRecorder [21], which captures the edit operations executed on Eclipse IDE [1]. Our tool is an extension of existing history refactoring tool named Historef [10, 11]. When OperationRecorder sends an edit operation to Historef [11], an associated appropriate leaf node is selected based on the information of the recorded edit operation. For example, when a refactoring operation is conducted, the caption of the node can be extracted from the target and the type of the refactoring operation.

Our tool is able to capture six popular refactoring operations that are provided by Eclipse IDE and that fit on the hierarchy structure: Rename Field, Rename Method, Rename Local Variable, Move, Extract Method, and Encapsulate Field. Also, we prepared some additional nodes. The first one is for representing the automated whitespace insertion and deletion, which are used for formatting source code. The second one is for representing the addition of new classes and methods. The last one is specialized for classifying the other changes that could not belong to any other nodes.

Figure 10 shows the resulting tree structures of the generated nodes. Its root node has four child nodes. The first child is the *refactoring*-node that represents the entire refactoring operations. The second child is the *whitespace*-node that represents the automated editing of white spaces. The third child is the *new-addition*-node that represents the creation of new classes and methods. The last child is the *etc.*-node that represents the other operations. The whitespace- and etc.-nodes are regarded as leaves.

The refactoring-node consists of several nodes that represent the type of refactoring operations in parent-child relationships. We also provided the rename-node that represents the entire refactoring about renaming since there are several types of renaming refactorings. Rename-node has three children: ones for Rename Field, Rename Method, and Rename Local Variable.

A screenshot of the implemented tool is shown in Figure 11. This screenshot shows a visualized hierarchy when



(a) Built hierarchy. (b) Reorganization of edit operations.

Figure 12: Application results.

Table 1: Numbers of Commits

Policy	# expected commits	# actual commits
A	22	22
B	12	12
C	3	3

applying a Rename Local Variable refactoring and the other non-refactoring changes. The conducted edit operations are automatically shown in the tree view during the editing process of the developer. This view also allows users to correct the tree structure. Users can fix the category of changes and to add new nodes when finding some mistakes about the categorization. The dividing commits feature can be invoked by selecting nodes from the given hierarchy.

5. APPLICATION EXAMPLE

5.1 Target

We have applied our technique for an edit history of a Java file included in the project developing a Twitter client for Android OS. The file includes 3903 lines of code, 29 of fields, and 105 of methods. One of the authors applied large refactoring and manual edits to the code because it had several maintainability problems such as a duplicated code fragment and the violation of the coding conventions on variable names. From the editing process during 40 min, we obtained an edit history consisting of 37 changes and 796 editing chunks. The applied refactorings include five of Rename Field, four of Rename Method, two of Rename Local Variable, four of Encapsulate Field, five of Extract Method, a formatting, and six of the other non-essential changes.

The source file were edited using the implemented tool explained in Section 4 so that we obtained the corresponding hierarchy. Figure 12(a) shows the obtained hierarchy. This hierarchy consists of 22 leaf nodes.

5.2 Results

We applied our technique to this history using the prepared three commit policies shown in Section 2 and obtained the divided commits for each policy. The dashed squares in Figure 10 indicate which nodes are captured by each policy. Policies A, B, and C are respectively regarded as fine-, medium-, and coarse-grained ones.

Figure 12(b) shows how the obtained edit operations were reordered and merged for each policy. In the figure, the color of changes represents the node to which it belongs and corresponds to the color of the node of the hierarchy in Figure 12(a). The number in a change is the total number of the leaf nodes to which the change belongs. The black arrows of the vertical direction between changes specify the non-trivial associations before and after the change of the node selection. The edit histories (1), (3), and (5) are respectively the ones according to the fine-, medium-, and coarse-grained commit policies before applying change reordering, whereas (2), (4), and (6) are the results after applying change reordering.

As a result, the change reordering for each policy succeeded without any conflicts, and we could obtain the appropriate changes for each policy. Table 1 shows the numbers of expected and actual commits for each policy. The number of the obtained commits was the same as that of the expected commits for each policy.

As shown in Figure 12, we can see that different commit policies produce different sets of commits. Policy A (fine-grained) produced the changes similar to the original ones. In Policy B (medium-grained), some changes were reordered, and the number of the produced changes then increased. Since Policy C (coarse-grained) is the coarsest-grained, the number of the reordered changes is less than that of Policy B (medium-grained).

5.3 Discussion

For the used commit policies of three types, it was possible to divide commits at different granularities. We confirmed that the change contents of each commit were consistent with the meaning indicated by the belonging node that constitutes the hierarchy. We also confirmed that the size of the obtained commits was suitable in accordance with each selected commit policy. These facts indicate that the proposed technique can divide commits if we use a refactoring-aware commit policy as mentioned above. Regarding refactoring operations, commit log messages obtained using Policy A were approximately equivalent to the actual editing process, which applies five different types of refactoring operations irregularly. This means that we can obtain the same sequence of commits if committing every after applying a refactoring operation. However, such flow forces developers to be inter-

rupted during their large refactoring process and is really time-consuming.

Policy B summarizes several renaming changes as a single commit. Therefore, Rename Field refactoring operations, which were distributed at second and thirteenth commits in Policy A, are located in close. This fact indicates that we are required to apply refactoring operations in the different order in order to manually obtain the same sequence of commits.

6. RELATED WORK

Study of tangled changes and techniques of untangling them are proposed. For example, Herzig *et al.* reported that 33.8% of commits are tangled in average [15]. Nguyen *et al.* also reported that 11–39% of fixing commits include other changes [20]. There are several types of untangling approaches including metric-based [6, 15], change template-based [17], and dependency-based [4,20]. Also, a proposal of an interactive environment for manipulating changes allows developers to reorganizing commits [3]. However, most of them do not focus on the preference of granularity of changes and hierarchical relationship of refactoring operations.

Techniques for detecting the instance of refactoring operations from changes are proposed [22,23], and detected results can be used for improving the understandability of source code differences [8, 13]. In our technique, we obtain the information of refactoring operations from IDE only, and the results of manual refactoring are missed. By analyzing the contents of changes, we can put the resulting edit operations of manual refactorings on our hierarchy.

Some refactoring-aware configuration management techniques [7, 14] capture and replay refactoring operations in order to make them portable. When reordering the changes in our technique, swapping of two changes fails if a textual dependency between them is found. Our approach is complementary to their approaches; using such replaying mechanism to our dependency checking might improves the applicability of our history reordering.

7. CONCLUSION

In this paper, we proposed a technique for reorganizing changes according to the refactoring-related commit policy that the target development project follows. In our technique, we record the edit operations of source code by developers and categorize them in a hierarchy based on the types of the operations. By selecting the nodes of the hierarchy as a reordering criterion and reordering the operations, we obtain the set of changes appropriate for committing according to the used commit policy. We have implemented a supporting tool as a plug-in of Eclipse, which enables us to build change hierarchy and reorganize changes based on the hierarchy. An application of our technique consisting of different kinds of refactorings was conducted to confirm whether we could obtain different sets of commits according to different commit policies. As a result, we could obtain different sets of commits following different granularities of policies.

The future work can be summarized as follows.

Evaluation. In this paper, we confirmed only whether appropriate changes are organized or not. We are planning to conduct a human study for checking the usability and the actual effort using our tool.

Richer hierarchy. The current types of node in our hierarchy are limited. We will define other types of nodes for categorizing edit operations in various way.

8. ACKNOWLEDGMENTS

This work was partly supported by JSPS Grants-in-Aid for Scientific Research (Nos. 15H02685 and 15K15970).

9. REFERENCES

[1] Eclipse. `http://www.eclipse.org/`.

[2] Git commit good practice - OpenStack wiki. `https://wiki.openstack.org/wiki/GitCommitMessages`.

[3] T. Barik, K. Lubick, and E. Murphy-Hill. Commit bubbles. In *Proc. 37th International Conference on Software Engineering* (*ICSE 2015*), pages 631–634, 2015.

[4] M. Barnett, C. Bird, J. Brunet, and S. K. Lahiri. Helping developers help themselves: Automatic decomposition of code review changesets. In *Proc. 37th International Conference on Software Engineering* (*ICSE 2015*), pages 134–144, 2015.

[5] S. Berczuk and B. Appleton. *Software Configuration Management Patterns: Effective Teamwork, Practical Integration.* Addison-Wesley, 2002.

[6] M. Dias, A. Bacchelli, G. Gousios, D. Cassou, and S. Ducasse. Untangling fine-grained code changes. In *Proc. 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering* (*SANER 2015*), pages 341–350, 2015.

[7] D. Dig, K. Manzoor, R. Johnson, and T. N. Nguyen. Refactoring-aware configuration management for object-oriented programs. In *Proc. 29th International Conference on Software Engineering* (*ICSE 2007*), pages 427–436, 2007.

[8] X. Ge, S. Sarkar, and E. Murphy-Hill. Towards refactoring-aware code review. In *Proc. 7th International Workshop on Cooperative and Human Aspects of Software Engineering* (*CHASE 2014*), pages 99–102, 2014.

[9] C. Görg and P. Weißgerber. Detecting and visualizing refactorings from software archives. In *Proc. 13th International Workshop on Program Comprehension* (*ICPC 2005*), pages 205–214, 2005.

[10] S. Hayashi, D. Hoshino, J. Matsuda, M. Saeki, T. Omori, and K. Maruyama. Historef: A tool for edit history refactoring. In *Proc. 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering* (*SANER 2015*), pages 469–473, 2015.

[11] S. Hayashi, T. Omori, T. Zenmyo, K. Maruyama, and M. Saeki. Refactoring edit history of source code. In *Proceedings of the 28th IEEE International Conference on Software Maintenance* (*ICSM 2012*), pages 617–620, 2012.

[12] S. Hayashi and M. Saeki. Recording finer-grained software evolution with IDE: An annotation-based approach. In *Proc. Joint ERCIM Workshop on Software Evolution and International Workshop on Principles of Software Evolution* (*IWPSE-EVOL 2010*), pages 8–12, 2010.

[13] S. Hayashi, S. Thangthumachit, and M. Saeki. Rediffs: Refactoring-aware difference viewer for java. In

Proceedings of the 20th Working Conference on Reverse Engineering (*WCRE 2013*), pages 487–488, 2013.

[14] J. Henkel and A. Diwan. CatchUp!: Capturing and replaying refactorings to support API evolution. In *Proc. 27th International Conference on Software Engineering* (*ICSE 2005*), pages 274–283, 2005.

[15] K. Herzig and A. Zeller. The impact of tangled code changes. In *Proc. 10th International Workshop on Mining Software Repositories* (*MSR 2013*), pages 121–130, 2013.

[16] D. Kawrykow and M. P. Robillard. Non-essential changes in version histories. In *Proc. 33rd International Conference on Software Engineering* (*ICSE 2011*), pages 351–360, 2011.

[17] H. Kirinuki, Y. Higo, K. Hotta, and S. Kusumoto. Hey! are you committing tangled changes? In *Proc. the 22nd International Conference on Program Comprehension* (*ICPC 2014*), pages 262–265, 2014.

[18] E. Murphy-Hill and A. P. Black. Refactoring tools: Fitness for purpose. *IEEE Software*, 25(5), 2008.

[19] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5–18, 2012.

[20] H. A. Nguyen, A. T. Nguyen, and T. Nguyen. Filtering noise in mixed-purpose fixing commits to improve defect prediction and localization. In *Proc. 24th IEEE International Symposium on Software Reliability Engineering* (*ISSRE 2013*), pages 138–147, 2013.

[21] T. Omori and K. Maruyama. A change-aware development environment by recording editing operations of source code. In *Proc. 5th Working Conference on Mining Software Repositories* (*MSR 2008*), pages 31–34, 2008.

[22] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim. Template-based reconstruction of complex refactorings. In *Proc. 26th International Conference on Software Maintenance* (*ICSM 2010*), 2010.

[23] S. Thangthumachit, S. Hayashi, and M. Saeki. Understanding source code differences by separating refactoring effects. In *Proceedings of the 18th Asia Pacific Software Engineering Conference* (*APSEC 2011*), pages 339–347, 2011.

The Driving Forces of API Evolution

William Granli and John Burchell
Computer Science and Engineering
University of Gothenburg
Gothenburg, Sweden
william.granli@gmail.com,
john.a.burchell@gmail.com

Imed Hammouda and Eric Knauss
Computer Science and Engineering
Chalmers | University of Gothenburg
Gothenburg, Sweden
{imed.hammouda,eric.knauss}@cse.gu.se

ABSTRACT

Evolving an Application Programming Interface (API) is a delicate activity, as modifications to them can significantly impact their users. The increasing use of APIs means that software development organisations must take an empirical and scientific approach to the way they manage the evolution of their APIs. If no attempt at analysing or quantifying the evolution of an API is made, there will be a diminished understanding of the evolution, and possible improvements to the maintenance strategy will be difficult to identify. We believe that long-standing software evolution theories can provide additional insight to the field of APIs, and can be of great use to companies maintaining APIs. In this case study, we conduct a qualitative investigation to understand what drives the evolution of an industry company's existing API, by examining two versions of the API interface. The changes were analysed based on two software evolution theories, and the extent to which we could reverse engineer the change decisions was determined by interviewing an architect of the API. The results of this analysis show that the largest driving force of the APIs evolution was the desire for new functionality. Our findings which show that changes happen sporadically, rather than continuously, appear to show that the law of Conservation of Organisational Stability was not a considerable factor for the evolution of the API. We also found that it is possible to reverse engineer change decisions and in doing so, identified that the feedback loop of an API is an important area of improvement.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, Enhancement—*Restructuring, reverse engineering, and reengineering*

Keywords

API Design, Software Evolution, Software Maintenance

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

IWPSE'15, August 30, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3816-5/15/08...$15.00
http://dx.doi.org/10.1145/2804360.2804364

1. INTRODUCTION

As the software industry and the open-source movement continue to grow, the number of public Application Programming Interfaces (APIs) is steadily increasing. APIs can improve the development speed [28], contribute to higher quality software [28] and increase the reusability of software [2]. There is a consensus in that modifications to APIs could negatively impact the users of the API [5, 14, 20, 23]. The main reason being that API users are required to update their application code, thus causing a disruption in the application's software ecosystem [21]. In the discipline of software evolution, such updates to software are studied from an evolutionary standpoint. Software can be updated for different reasons and these motives can be grouped into corrective, adaptive, perfective and preventive changes [19].

Modifications to deployed APIs may negatively impact its users. It is therefore important to investigate why these changes occur. Understanding the motives behind the changes could help API architects to prevent potential future changes to their APIs. Before industry can safely include such information in their feedback loop, we must critically assess to what degree we can reverse engineer such motives. Identifying the types of changes that occur when an API evolves will improve communication and discussions regarding maintenance and evolution tasks for the case company. This improved communication could provide a common language that developers, users and management can use to discuss potential evolution of the APIs. Understanding the evolution theories could further assist understanding of how change will occur and how to plan for it. Lastly, by keeping a history of changes to an API over time, trends can be identified which could help with future API design.

Previous studies have investigated programming language APIs, libraries or frameworks focusing on causes and effects of modifications to APIs [10, 15, 27]. However, to the best of our knowledge, no studies have been performed on embedded platform APIs from a software evolution perspective. Our study will fill this gap by analysing an embedded platform API, classifying the types of changes that occur and analysing them with the use of software evolution theories.

The goal of the study was twofold. Firstly, the aim was to explore what are the driving forces of API evolution and secondly, to validate the results' level of correctness by comparing our findings with the case company's explanation of the changes. With this information, we could determine to what extent these change decisions can be reverse engineered.

The sections following the Introduction are structured as follows: In Section 2, we present the background and previously published work that is related to our study and introduce the case company and their API. A description of our methodology is introduced in Section 3. In Section 4, the results of our study are presented and in Section 5, we discuss the implications of our work. In Section 6 we summarise the study and give suggestions for future work that builds on our research.

2. BACKGROUND

This section will introduce the fields of APIs, software evolution and it will provide a review of studies that are related to our study. The subsequent section will introduce the case company with a description of the API used in this study.

2.1 APIs

An API offers an interface through which developers can access programmatic functionality. APIs can be seen as having three interacting layers; the public interface, the actual implementation of the functionality and an intersecting layer which provides utilities such as error handling, third party libraries and additional auxiliary features. Typically, the interfaces provide the definitions of functions and data structures while the implementation realises those interfaces.

API design is notoriously difficult, as a myriad of design and performance decisions must be taken into consideration when creating APIs [2, 6, 28]. Examples of such design decisions include how to structure an object's constructor parameters or if the API should display errors at compilation or at runtime [28]. More trivial design problems, such as assigning names to API features or correctly naming user-defined types, can have a significant impact on the usability of an API [27]. When facing such design challenges, the following four factors are important to consider: a) The API must be understandable through good documentation, b) the API must not be overly abstract, c) the API must be reusable and d) the API must be easy to learn [27]. One of the most important qualities in an API is that the intent of the API must be clear to the user [27, 28]. The design decisions reached during development of an API will affect the overall usability of the API. Measuring such an effect can be done by investigating the twelve cognitive dimensions that are impacted by interactions between the API and its users [8].

2.2 Software Evolution

Software evolution is a field that studies the application of software maintenance activities, changes in software processes and the resulting, evolved versions of the software. The concept of software maintenance has existed since the 1960s when it was first introduced to the software development community [19].

A set of four categories describing different kinds of software maintenance [19] became the basis upon which twelve new types were developed [7]. The twelve types of software evolution and software maintenance, as seen in Table 1, describe a software evolution activity that relates to one of three particular areas; the code, the software and the customer-experienced functionality.

Type	Explanation
Enhancive	Inclusion of new functionality
Corrective	Corrections of functionality
Reductive	Reductions of functionality
Adaptive	Inclusion of new technology
Performance	Improvements to performance
Preventive	Improvements to future maintainability
Groomative	Improvements to maintainability
Updative	Updates to documentation
Reformative	Changes to documentation
Evaluative	Inspection activities
Consultive	Consultations activities
Training	Training activities

Table 1: Types of Software Evolution [7]

1. **Continuing Change**
   E-Type systems must be continually adapted else they become progressively less satisfactory.

2. **Increasing Complexity**
   As an E-Type system evolves its complexity increases unless work is done to maintain or reduce it.

3. **Self Regulation**
   E-Type system evolution process is self regulating with distribution of product and process measures close to normal.

4. **Conservation of Organisational Stability**
   The average effective global activity rate in an evolving E-Type system is invariant over product lifetime.

5. **Conservation of Familiarity**
   As an E-type system evolves all associated with it, developers, sales personnel, users, for example, must maintain mastery of its content and behaviour to achieve satisfactory evolution. Excessive growth diminishes that mastery. Hence the average incremental growth remains invariant as the system evolves.

6. **Continuing Growth**
   The functional content of E-Type systems must be continually increased to maintain user satisfaction over their lifetime.

7. **Declining Quality**
   The quality of E-Type systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes.

8. **Feedback System**
   E-Type evolution process constitute multi-level, multi-loop, multi-agent feedback systems and must be treated as such to achieve significant improvement over any reasonable base.

Table 2: Lehman's Laws of Software Evolution [18]

Software evolution theory suggests that software programs can be grouped into three different types of systems, namely S-programs, P-programs and E-programs [17]. S-programs are programs which are exactly derivable from a specification. An example of such a program is one which finds the lowest common denominator for two integers. A key characteristic of an S-program is that if its functionality is changed, the end product will be a new program and not an evolved version of the previous program, since the mapping between specification and program is exact. P-programs are programs which are inherently more complex, and "solve a problem which can be precisely formulated, but whose solution must inevitably reflect an approximation of the real world". Furthermore, P-programs are programs which are centred around the perception of users, analysts or programmers. This perception is often commonly a mathematical model of the problem. An example of such a program is a program which predicts weather, since it is derived from a set of hydrodynamic equations. E-programs are described as "inherently even more change prone [than S-programs and P-programs]. They are programs that mechanize a human or societal activity". These factors can be explained by the difference between P-programs and a E-programs which is that E-programs "change the very nature of the problem to be solved" by "becoming a part of the world it models". Further, a characteristic of E-programs is that, during development, the specification must be developed by involving predictions of the consequences of the deployment of that system will have, since once the program is deployed "questions of correctness, appropriateness, and satisfaction arise which inevitably leads to additional pressure for change". Lehman's eight laws of software evolution are said to apply to such E-programs. The laws are presented in Table 2.

2.3 API Evolution

The analysis of APIs in the context of software evolution has primarily focused on APIs that are part of large programming languages, such as Java [15, 27] and Smalltalk [23]. Attempts to uncover the intents of the changes made in the Java libraries, AWT and Swing, have also been undertaken [15], leading to suggestions that the use of a strong architecture is vital for ensuring the successful evolution of an API [15]. Investigations of three frameworks and one library indicated that 80% of refactoring changes to APIs negatively affected existing applications [10]. The changes, sometimes referred to as ripple effects [23] indicate that negative effects of changes can propagate throughout a software ecosystem. It was found that 14% of non-trivial API deprecations caused errors in at least one project, with the worst case of 79 projects being affected [23].

2.4 Case Company Description

The case company of this study will from here on be referred to as *company A*. Company A operates in the domain of video surveillance and has offices in 49 countries worldwide. Company A is the global market leader in the markets of network cameras and video encoders. They develop embedded software for security cameras. The cameras are designed to be accessible through APIs that are developed and maintained by the company.

The API analysed in this study is written in the programming language C and has been deployed for several years. It was recently updated from v1.4 to v2.0, which is the current active version. The changes between these two versions are what has been analysed in this study. Between the release of v1.0 and v1.4, no functionality was added, as it only added additional build options for other hardware platforms. The API is used by company partners, which are other companies developing applications for the cameras. These applications are, in turn, used by the end-users of the cameras. The API is compatible with a wide range of camera types that offer different functionality and are used for different purposes.

2.4.1 APIs as Systems

Company A's API is rather complex, and in addition to the API interface and implementation code, it includes a middle-layer which manages error handling and the access to various 3rd party libraries. In this study, we use the API interface and the middle-layers as units of analysis. Examples of where the responsibilities exceed those of a primitive API interface is that it includes functionality such as error handling and a daemon which facilitates communication between hardware entities. In addition to that is an SDK which includes the API documentation, user manuals and example applications that utilise the API code. Based on these characteristics and the IEEE definition of a system, which reads as follows; "A combination of interacting elements organised to achieve one or more stated purposes...Organised to accomplish a specific function or set of functions within a specific environment" [1], we consider the API to be a system because of the combination of the interacting parts of the API, coupled with the specific nature of the problem the API is addressing.

Building on this, we regard the API examined in this study to be an E-type system based on that it is a part of the world it models. This is evident from that it is a vital part of the cameras' and the only option for developing software for the cameras. The API is also considered by company A to be a vital part of the cameras' software ecosystem. This factor corresponds with the characteristic of E-type systems which relates to the system being susceptible to change from pressure that occurs after the system has been deployed. Similarly, we do not consider the API to be a P-type system, based on the fact that the system is virtually impossible to model due to its high dependence on the environment in which it operates in.

3. METHODOLOGY

This study has been conducted using the case study methodology [25]. Our research can be classified as an embedded case study [30], since both the API interface and API documentation have been used as units of analysis. The reason the selected methodology was used is that it is essential to study the phenomenon of API change in its natural context. The applicability of case studies in such scenarios is supported by existing literature [4, 24, 25, 30]. An alternative approach that was considered was design research, but if a prototype API was to be used instead of one which is tried and tested in an industry setting, the study would lack real-life context [25]. An additional motivation for why the case study approach was used is that there is little existing research conducted in the area of drivers of API change and that input from the industry is vital to the success of the research.

Phase 1: Data Collection

1. Inspect v1.4
2. Inspect v2.0
3. Identify changes between versions
4. Conduct interview
5. Structure gathered data
6. Formulate hypotheses

Phase 2: Data Analysis

1. Code changes
2. Group changes
3. Categorise groups
4. Identify and formulate trends
5. Analyse based on types [7]
6. Analyse based on laws [18]
7. Validate hypotheses

Table 3: Methodology Overview

The study was conducted in two phases. The aim of the first phase was to generate hypotheses and to formulate clear research questions. This was achieved by using a data-driven approach to analysing the interface code and documentation. The second phase included analysis of the interviews, during which we aimed to confirm the hypotheses that were formulated in the first phase. Both phases were performed iteratively. This created the opportunity to improve the data analysis as the study progressed, as well as, allowing us to adapt to possible changes in direction, due to the hypothesis generating approach of the first phase. An overview of the methodology can be found in Table 3.

The process used for examining the code and documentation was inspired by grounded theory analysis [26], as it is recommended for hypothesis generating studies [25, 26]. The exploratory analysis lead us to formulate the following research questions.

RQ1 What drives API evolution?

RQ2 To what extent can we reverse engineer API change decisions?

3.1 Data Collection

The data were collected primarily through inspections of the source code and of accompanying documentation. The source code was comprised of the two API versions previously mentioned. Each version of the API had its own respective documentation, including example code, a library description and a basic development manual.

The API interface code was inspected sequentially, starting with v1.4. Both versions of the API underwent the same type of inspection. This involved extracting method signatures, enums, structs, typedefs and macros, which were subsequently stored in spreadsheets. The extraction involved a lot of manual work, but Git's [12] diff command was used to identify changes in files which exist in both versions. After a change was identified in the code, a description of the corresponding change in the documentation was added to the spreadsheet. The module and file in which the change was identified was then logged together with an ID and a description of the change. The data collection from code and documentation was performed independently by both researchers to reduce the risk of human errors affecting the results. If the data collected by each researcher differed, the cause of the discrepancy was investigated and resolved.

After the initial source code and documentation inspection, an interview was conducted. The main purpose of the interview was to validate our findings from the code inspection with the API architect. The interview also served the purpose of providing additional insight into what drove the evolution of the API, by complementing what was found from the inspections. The final reason for interviewing the API architect was to gather information about the API users and how their decisions, needs and requirements have affected the development of the API.

The interview was conducted using a semi-structured approach [24]. Structure to the interview was provided by organising it around the software evolution theories [7, 18]. Investigative and open-ended questions related to each type and law were asked to provide a basis for comparison between the answers and our initial analysis. The interviewee was encouraged to speak freely, even if it required a change in direction or topic. This was encouraged in order to fulfil our exploratory goal of the study.

3.2 Data Analysis

To analyse the data, each change identified in the spreadsheet was coded based on which module the change occurred in and based on if the change added, removed, modified or did not affect functionality. Grouping the codes by module allowed us to form general concepts of change. However, changes that affected multiple modules were classified on their own. These concepts were then grouped into categories which were based on common patterns between the concepts. An example of a pattern found during the analysis was that several concepts followed the pattern of adding new functionality. These categories were then used to identify the most notable trends in the evolution of the API. These trends were later used as a basis for discussion during the interview.

The trends identified previously were then classified with the help of the decision tree used for classifying the types of software evolution and software maintenance [7]. Each of the previously identified trends were given a main type of change and, if appropriate, a secondary type. Main types of change identify the primary motive for a trend, whereas the secondary types are additional motivations for a trend. For example, if an identified trend was the re-implementation of an old module that included new functionality and made use of a new interface, it would be classified as mainly enhancive and secondarily groomative. In addition to this, we analysed the trends based on Lehman's Laws, to determine if they applied to the changes made to the API. This information, as well as the results of the interview, were then used as the

basis for answering **RQ1**.

To allow us to answer **RQ2**, we compared our analysis and interpretation of the trends identified prior, against what was expressed by the API architect, during the interview. Success was determined by comparing the inter-rater reliability, by calculating Cohen's kappa [9] for the gathered data.

3.3 Validity Threats

In this section we discuss possible threats to construct validity, internal validity, external validity and reliability [25].

Construct validity Since our study is largely based on existing theories [7, 18], the validity of it directly correlates to the validity of the theories and the applicability of them. With regards to Lehman's laws of software evolution, the laws are said to only apply to E-type systems. According to our analysis, we consider the API to be of the E-type, but if our interpretations of Lehman's definitions are incorrect, it could be considered a threat to the construct validity of our study.

Internal validity Limitations related to internal validity have been acknowledged by analysing the code and documentation jointly. This has contributed to triangulating the results and discovering possible inconsistencies. Further triangulation of the results was performed by interviewing an architect of the API. Business-related factors might have had an impact on the evolution of the API, and these were not closely investigated. This was not in the scope of the study and would best be investigated in a study complementary to ours. The study revealed that the API users were significant to the evolution of the API. Since API users were not interviewed, it might be an affecting factor which was left unexplored.

External validity Since only two versions of the API were analysed, we cannot claim any generalisability for our study. We do, however, believe that our study can offer comparability for companies that chose to do a similar analysis on their own APIs.

Reliability Due to restricted access to company A's source code, we were not able to include versions prior to v1.4 in our analysis. This means that the evolution of the API was studied during a rather short period of its lifetime. It was also not possible to analyse the implementation of the API, something which might have contributed to more accurate predictions of the motives behind the changes. We consider these factors to be the remaining threats to reliability. Reliability has been increased by conducting the study according to an accredited guide to case studies [25]. In addition to that, strategies from additional well-established papers [3, 24, 26] have been used to increase the trustworthiness of the data gathering and data analysis.

4. RESULTS

This section presents the results of our study and is structured around the research questions. Each section begins with a summary of the results and a subsequent in depth description of the findings.

In total, thirteen unique changes were identified when analysing the source code, the majority of which were additions of functionality of the API. Such additions include added functionality to allow mechanical and digital control of the camera, to utilise additional storage devices, to allow serial connectivity and to allow audio analysis on the cameras. Changes that were pure additions to the API were not attributed with any secondary types of change.

Type	Main Types	Secondary Types
Enhancive	7	-
Corrective	-	-
Reductive	1	-
Adaptive	1	-
Performance	-	-
Preventive	2	-
Groomative	-	5
Updative	1	1
Reformative	1	1
Evaluative	-	-
Consultive	-	-
Training	-	-

Table 4: Identified Types of Change

Many of the changes were deprecations that were later re-implemented as new modules. Changes of this kind were grouped into a single type instead of two. Such examples include the deprecation and re-implementation of an event system, dynamic web page generation and configuration utilities. Changes of this kind often had an accompanying secondary type of change.

The remainder of the changes were defined as general trends. These changes were not changes to individual parts of the API, but were instead broader changes. Such examples include a shift towards using interfaces to access features, a different error-handling approach and the inclusion of additional 3rd party libraries. Typically, these general trends also have a secondary type of change attributed to them.

4.1 The Driving Forces of API Evolution

The results show that certain types of change are more prevalent than others, these results are shown in Table 4. Analysis of the results showed that enhancive changes were the primary types of change that drove the evolution of this API and that they acted as a catalyst for many of the other changes in the API. Updative changes were found to be consequences of enhancive changes, rather than being drivers of evolution themselves. Supporting the users of the API through the combination of preventive and groomative types of change, together formed another strong driver of the API's evolution. Reductive and adaptive changes, while important, did not strongly drive the evolution of the API. No evidence was found for corrective or performance types of change. Similarly, there was no evidence found for evaluative, consultive or training activities.

Our analysis of the results related to **RQ1** showed that seven out of the eight of Lehman's laws [18] are present in the API. A summary of our findings can be seen in Table 5. The evidence found for the laws of Continuing Change, Self Regulation, Conservation of Familiarity, Continuing Growth and Feedback System mainly relate to enhancive changes that add functionality to the API. It was also found that the laws of Increasing Complexity and Declining Quality do

Law	Applicability
Continuing Change	True
Increasing Complexity	True
Self Regulation	True
Conservation of Organisational Stability	False
Conservation of Familiarity	True
Continuing Growth	True
Declining Quality	True
Feedback System	True

Table 5: The Applicability of Lehman's laws

apply, but that the changes which support these laws usually had a secondary intent related to complexity or quality. The only law which was found to have little effect in the evolution of the API analysed in this study was the law of Conservation of Organisational Stability.

Following this section, we will discuss the results related to each type of change category, discussing the results for each of them with examples from the changes identified between the two versions of the API. Subsequently, each of Lehman's laws will be discussed and our results will be presented showing if the laws were observed.

4.1.1 Types of Change

Enhancive Additions or re-implementations of functionality are defined to be enhancive changes. In addition to this, we have included deprecations that have then been re-implemented to also be enhancive. Seven of the identified changes were given the type of enhancive. The majority of which were additions of new functionality to the API. Two examples of added functionality are the additions of a camera movement module and an audio analysis module. One example of a deprecation that was re-implemented is the event system. Given that over half of the changes were identified to be enhancive, we conclude that the APIs evolution primary type of change was enhancive.

Corrective Corrective changes are identified as types of change that fix broken functionality. No corrective changes were identified between v1.4 and v2.0. There are two main reasons that we did not find any changes of this type. Firstly, the majority of changes were additions or re-implementations of functionality. This meant that most of the existing code was either removed, replaced or deprecated, resulting in little to no obvious fixes. Secondly, we only inspected public facing API and not the implementation itself. We therefore conclude that the evolution of the API was not driven by corrective changes.

Reductive Reductive changes are defined as changes that remove or reduce functionality. This is distinctive from a deprecation as reductive types of change remove functionality completely. Only a single reductive type of change was identified. This reductive change removed functionality that provided the ability to buffer individual images for closer analysis by a user. Given that it is more common to refactor APIs and not remove functionality [10, 29], this result was not surprising. It was revealed during the interview that the functionality was removed because company A no longer wished to support this feature. As this was the only reductive change found, we conclude that the evolution of the API was not primarily driven by reductive changes.

Adaptive Adaptive changes are defined to be changes which involve changes to technology or resources used. A single change was identified as being adaptive; the inclusion of new 3rd party libraries. Libraries were added for audio and data communication with the pre-existing low-level library being incorporated into the API. We therefore conclude that the evolution of the API has not been primarily driven by adaptive changes.

Performance Performance changes are defined to be types of change that intentionally alter system performance. No evidence of performance-driven changes were identified in the API. This therefore leads us to conclude that the API's evolution is not driven by performance changes.

Preventive Preventive changes are defined to be changes that attempt to avoid future maintenance. This should not be confused with changes that attempt to make the current state of the system more maintainable. Two changes were identified to be preventive types of change. A new error handling system was one of these changes. v2.0 of the API added specific error handling modules and functions which replaced the pre-existing error-handling in v1.4 of the API. The redesign of the system to make use of interfaces was also classified as a preventive change. Using interfaces to access modules helps to make them more maintainable and more easily extensible. Preventive types of change appear to drive the evolution this API more strongly than other types of change. This could be due to the fact that as the API grows, the need for higher maintenance also increase, a fact that also is supported by Lehman's laws.

Groomative Groomative changes are defined to be changes that aim to immediately make the code more maintainable. While none of the changes identified were primarily groomative changes, three of them were secondarily classified as being groomative. This indicates that the groomative nature of a change is often a positive affect of other types of change. The move towards using interfaces can be seen as an example of this. While the primary motive of change was to make the API more maintainable in the future, making the API more maintainable immediately was also a reason for the change. While not a primary factor for any of the changes, groomative types of change should still be viewed as important to the API's evolution.

Updative Updative changes are defined to be changes to documentation of a system that are made to conform with changes in the code. A single change was determined to be updative; updating the API documentation to include the new functionality of v2.0. The updates included a new API specification, deprecation list and the inclusion of the new features and libraries. Updative changes have very little impact on the functionality of the API yet they are still important changes for the usability of the API [27]. Changes to the API cause the need for updative changes to its documentation, without them, the usability of the API could suffer. We therefore conclude that the evolution of this API was not driven by updative changes.

Reformative Reformative changes are defined to be changes that update the documentation to the stakeholders needs. An example of such a change is that the structure of the documentation is changed. One reformative change was identified between v1.4 and v2.0; a redesign of the existing example programs. The aim of this change was to update the examples to illustrate the new functionality in the API, showing how the modules interact and at the same time, presenting a standard for how to use the API. We therefore conclude that reformative changes, while important for the usability of the API, did not drive the evolution of the API.

Evaluative, Consultive and Training Evaluative, consultive and training are all types of activities, rather than changes that a system can undergo. Evaluative activities are defined as activities such as auditing or evaluating the software. Similarly, consultive activities involve consultations with customers about the software. Finally, training activities involve training for customers and users of the software. None of the aforementioned activities occurred as part of the APIs evolution. We can therefore conclude that these change types were unimportant for the evolution of the API.

4.1.2 Lehman's Laws

Continuing Change Company A described that including new functional content was one of the top priorities for the release of v2.0. This is supported by the characteristic of the changes identified in this study. Four of the thirteen changes were additions of new modules, and additional functionality was also added to re-implemented modules. It was also mentioned that the API users had previously requested more frequent updates to the API, despite the drawbacks this could cause. One of the main reasons for continuously adapting the API is also to control the way API users use the API. Prior to v2.0 being released, a number of API users used workarounds to implement applications which contained functionality that was not yet offered by the API. These applications were fully functional, but there existed no standard method of implementation. We therefore conclude that the law of Continuing Change was a driving factor for this API.

Increasing Complexity A majority of the changes introduced in v2.0 aimed to improve the structure and the way that the API was used. Many of these changes were related to the same design choices, which enforced a certain implementation style on the whole API. One example of this is the change to the interfaces which set a standard for the whole API. During the interview it was also concluded that consistency and following certain standards is an important factor to consider when designing APIs. It was also made clear that a contributing factor for changing certain parts of the API was to bring it into line with the new interface design. The interviewee mentioned that one way of achieving this was to follow guidelines or design best practices, but that company A currently did not make use of these. Although the importance of reducing the complexity was stated, it was made clear during the interview that one module was not updated between the versions, even though it did not follow the new standards of v2.0. In conclusion, we establish that the law of Increasing Complexity was a driving factor for this API.

Self Regulation The release of v1.4 did not include any changes related to the API interface. v1.0 through to v1.4 strictly added support for additional build target platforms. When comparing the changes introduced in v2.0 to the ones in previous versions, it is clear that the growth of the API has gradually increased. During the interview it was also mentioned that the future plans for the API did not include any major changes and that only a few minor changes were planned in the near future. The future updates to the API will also be made incrementally and there will not be a big bang update similar to that of v2.0. This supports that the law of Self Regulation was significant in the API's evolution, assuming that v2.0 can be considered to be the peak of the normal distribution curve of the APIs growth.

Conservation of Organisational Stability The changes to the API have been made very sporadically, where only two new versions have been released since the deployment of v1.0. As mentioned in Section 4.1.2, the changes introduced in v1.4 did not involve any changes to the existing API functions. Consequently, the only significant update to the API was made in the transition to v2.0. The internal development of the API has also been made sporadically and there have been periods where the API has not been actively developed. This suggests that the law of Conservation of Organisational Stability was not a considerable factor in the evolution of the API.

Conservation of Familiarity The documentation that is included with the API was rigorously updated with the introduction of v2.0. All functionality which was added or modified in v2.0 was updated accordingly in the documentation. The same pattern can be seen for the example code, which went through major changes as a result of the changes to the API interface. Company A's intent to control the way that their API is used also suggests that the knowledge of how to use the API is of importance, since if the API is used in too many different ways, it will be more difficult for company A to have relevant documentation and example code. It was also expressed during the interview that further transparency in how the API should be used could be achieved. One example mentioned of how this could be achieved is by improving the expressiveness regarding which API functions are supported on which hardware types. These facts support that the law of Conservation of Familiarity played an important part in the evolution of the API.

Continuing Growth Large parts of the changes between the versions were strictly additions of new functional content. 4 new modules were added and significant additions of new functionality were added to the modules which were re-designed from v1.4. The interviewee also made it clear that increasing the functionality was one of the main goals with the 2.0 update. This also ties in with the ambition of controlling the way that the API is used, as mentioned previously. It was also mentioned that company A had received feedback from the API users that more frequent updates to the functional content was desirable, even though this may require them to update their application code. We therefore conclude that the law of continuing growth was a driving force for the evolution of the API.

Declining Quality The interviews established that a groomative motive was not the primary reason for implementing any change. The changes to the example code and the documentation, as well as the re-designed modules are examples of these, where the main motive behind the change was to either prevent unwanted usage or to increase the functional content. These changes did, however, also increase the qual-

ity and were a substantial part of the update. Our analysis also shows that changes related to quality are bound to happen, as other types of changes often incorporate quality related aspects. We therefore settle that the law of Declining Quality was a driving force for the evolution of the API.

Feedback System One of the future goals of improvement of company A is to better include the API users and end-users of the applications in their feedback loop. Currently, only the API users are included in this feedback loop and the API architect expressed the desire to also include the end-users. The reasoning for this was to increase the quality, and to be able to deliver content which is useful for the users and which leads to good applications being developed. Prior to v2.0 being released, an example of how the API users were included in the feedback loop was when they implemented functionality related to sound, despite the fact there was no interface for this. The experiences of this API user, were later considered when developing the axsound module. One benefit of having a more structured feedback loop, would be that the API developers would have greater awareness of potential errors in the implementation of the API. Another ambition which was mentioned, was to bring scientific theories and best practices from research into consideration when developing the API. This would aim to complement the current strategy which is largely based around the expertise and experiences of the API architects. Based on these factors and clearly expressed ambitions, we conclude that the law of Feedback System was of significant importance during the evolution of the API.

4.2 Reverse Engineering Change Decisions

After cross-examining our initial analysis with the interviewee's responses, we found that 85% of our main type predictions matched the actual motive of the change and that 85% of our secondary type predictions were correct. The kappa value [9] for the predictions was 0.812, which shows that there is a high level of agreement between the predictions and the change decisions. In Table 6, a detailed view of each prediction juxtaposed against the actual change decision can be found for both the main and secondary types.

The changes related to functionality being added, modified (deprecated) or removed were predicted with a 100% accuracy. These changes were deemed the most significant by the interviewee, and it was expressed during the interview that the these types of changes were the main driving factor of the development of the new version. Contrary to this, three out of five changes which were unrelated to functionality were accurately predicted. The two changes which were not predicted accurately were inaccurate because the main type was confused with the secondary type. This shows that, although the prediction was incorrect, the general reasoning was correct. The changes which had no secondary type were all predicted accurately, which further shows that changes with several types can be diffuse.

5. DISCUSSION

RQ1: **What drives API evolution?**

Our results show that the major driving force for change in the case company's API was the call for increased functionality. The classification of the types of software evolution and software maintenance showed that enhancive changes

Main		Secondary	
Prediction	**Actual**	**Prediction**	**Actual**
Enhancive	Enhancive	N/A	N/A
Enhancive	Enhancive	N/A	N/A
Enhancive	Enhancive	N/A	N/A
Enhancive	Enhancive	N/A	N/A
Enhancive	Enhancive	Groomative	Groomative
Enhancive	Enhancive	Groomative	Groomative
Enhancive	Enhancive	Groomative	Groomative
Reductive	Reductive	N/A	N/A
Adaptive	Adaptive	N/A	N/A
Preventive	Preventive	Groomative	Groomative
Updative	Updative	Reformative	Reformative
Updative	Reformative	Reformative	Updative
Groomative	Preventive	Preventive	Groomative

Table 6: Categorisation of Changes

which either aimed to add modules or new functionality to existing modules were the most common. We also think that it is interesting to note that a majority of the changes which did not affect functionality, such as the updative changes, were spurred because of previous changes that were related to functionality. Following this line of thought, the desire for new functionality can be seen as a catalyst that allows new change to occur. The view of considering functionality to be a major driver for change is supported by the analysis with Lehman's laws. In the law of Continuing Change, Lehman concludes that "an E-type system must be continually adapted or it becomes progressively less satisfactory". The results of our work support this view, as the changes did not only add functionality, but it was also found that the application developers most major concern were updates to functionality.

In relation to this, previous work has identified that refactorings make up a substantial part of the changes APIs undergo [10, 11, 13, 29]. These results are not in line with ours, since our results show that groomative changes are uncommon and that changes related to quality are not primary drivers of change, rather, they are ripple-effects of changes to functionality [23].

The results also showed that usability was the second-most important driver of the studied API's evolution. The classified updative, groomative and reformative changes all affected the maintainability and usability of the API and involved activities that have been found to improve the usability and maintainability of an API [2, 8, 22, 27]. The interviewee discussed how company A wanted to aid developers creating applications by improving the interfaces that the developers are using. This improvement to the interfaces would better communicate the design and intentions to the developer. Improving the documentation of the API was also important for v2.0, as were improvements to the example code. These updates were intended to aid developers and create a set of standards that should be followed when creating applications. These changes correspond with three of Lehman's laws: Conservation of Familiarity, Declining Quality and Feedback Loop.

Our results found that seven out of eight of Lehman's laws affected the evolution of the studied API. The law of Conservation of Organisational Stability was found to have less significance, since changes to APIs are unfavourable due to the potential disruption the applications using the API could face. This inertia to change in API evolution is supported by other research, which also conclude that the negative effect that changes to APIs may have is a major factor to be considered [5, 14, 20, 23].

Based on our results, we form the conclusion that the desire for functionality and the reluctance of changing are two opposing forces that caused updates to this API to be infrequent. In the evolution of APIs, the negative effects of changes are greater than in other types of systems, which is the main indication that the law of Conservation of Organisational Stability is less significant for this API. This occurrence has been considered by Lehman, as he noted that "the laws are not immutable, since they arise from the habits and practices of humans" [17]. The underlying reason for this is that the laws are influenced by conscious thought processes, based on human understanding [16], which, in the case of this API, is the conscious decision to limit updates as much as possible.

RQ2: **To what extent can we reverse engineer API change decisions?**

We conclude that we were able to reverse engineer the change decisions to a large extent. This is clear from the results, which showed that 85% of the changes were reverse engineered accurately. The kappa value of 0.812 further supports this. We interpret that the significance of this result suggests that reverse engineering of change decisions can be done in situations where a perfect result is not required. In cases where a near-perfect result is enough, we do, however, believe that reverse engineering change decisions is worthwhile. This line of thought is supported by our results, which, in summary, show that we are able to reverse engineer the most major trends with very high accuracy, but that certain nuances of the changes can be misinterpreted.

5.1 Implications for Research

The implications of our findings should encourage research to further study APIs by relating them to existing software evolution theories. Considering that we were able to successfully reverse engineer the changes made to the API in this study, it opens up possibilities to base future research on, for example, the types of software evolution and software maintenance. Since the kappa analysis showed that there was a high level of agreement between our classification and the actual motives to the change, it shows that the types of software evolution and software maintenance could even be used as the basis for a large-scale quantitative analysis to map the types of changes made to APIs.

By analysing the API based on Lehman's laws, we were able to find that the law of Conservation of Organisational Stability had little impact upon the API investigated in this study. Our results show that the opposite of what the law suggests, in its current form, is true for this API. However, since only a single API was studied, we cannot generalise this finding for all APIs. Research should therefore be wary of assuming that this law is true for APIs and future work could be undertaken to fully determine its validity in regards to APIs.

5.2 Implications for Company A

The changes introduced to the API are mainly based on the expertise and experiences of the API architects at company A. There has not been a scientific approach to maintaining the system and the knowledge from research has not directly affected the way that the API's evolution has been managed. With our work, we hope to provide further insight into how the API is maintained. Our analysis of the API based on Lehman's laws should further assure that company A is moving the evolution of the API in the right direction. Our research proposes a dilemma for company A in how to tackle challenges related to the law of Conservation of Organisational Stability. Determining if the benefits of continuously and frequently updating the API outweigh the side-effects that these updates could have on the application developers, is something that company A will need to consider.

One way of finding the solution to this problem would be by in a more structured way including the application developers in the feedback loop. Since the application developers are on the receiving end of the changes, we believe that their needs have to be assessed before the problem can be solved. With our study we identified both that the law of Feedback System is of great importance to company A and that it is one of the key areas of API development that company A wishes to improve. One of the challenges posed by software evolution is that the activities carried out are often intangible and non-objective [7, 19]. Based on this, we propose that our study which has granulated the activities is used as a basis for communicating the current stage of the evolution of their API. We hope that this classification can be used in its current form, to help quantify the activities and to provide an overview of them. We also hope that the approach we used can be used to predict which types of activities should be emphasised in the future evolution of the API. With our study we have also demonstrated that the classification of changes is achievable, which, in turn, should encourage company A to adopt this method of classifying change.

6. FUTURE WORK

With this study, we conclude that studying API evolution with the help of existing theories, provides a new and structured approach to the field of API evolution. Our study shows that functionality is the largest driving force of change in the evolution of the studied API. It also suggests that changes to APIs occur infrequently and sporadically. This indicates that the law of Conservation of Organisational Stability did not play a significant role in the evolution of this API. However, given that this study was only performed on a single API, we cannot generalise this finding. Furthermore, we have identified the importance of having a structured feedback loop when maintaining an API. We have also provided company A with a foundation for facilitating such a change.

To build on our study, we propose that a study on the ecosystem's role in the evolution of APIs is conducted. We believe that this area could be a major source of improvement for API evolution. Exploring the benefits of including application developers and additional entities of the eco-system in the feedback loop, is a good starting point for company A. In such a study, we would suggest that the types of soft-

ware evolution and software maintenance [7] could be used as a mean of facilitating discussion between the entities. It could also be possible to study the effect that changes to APIs have on the application code, in such a study.

To further investigate what drives API evolution, we believe that API evolution should be studied from a business point of view. We therefore suggest that a similar study is conducted, but with the goal of analysing the change of an API based on business motives. Positioning a study around business motives would help with identifying the changes before they are formulated into technical changes, and it would provide a new and fresh angle to the field of API evolution.

Lastly, based on our successful attempt to reverse engineer the changes made to this API, we suggest that a quantitative study is conducted to investigate the distribution of the types of software evolution and software maintenance and the applicability of Lehman's laws on a large scale.

7. REFERENCES

[1] Systems and Software Engineering – Vocabulary. *ISO/IEC/IEEE 24765:2010(E)*, pages 1–418, Dec 2010.

[2] L. M. Afonso, R. F. d. G. Cerqueira, and C. S. de Souza. Evaluating Application Programming Interfaces as Communication Artefacts. *System*, 100:8–31, 2012.

[3] C. Andersson and P. Runeson. A Spiral Process Model for Case Studies on Software Quality Monitoring - Method and Metrics. *Software Process: Improvement and Practice*, 12(2):125–140, 2007.

[4] I. Benbasat, D. K. Goldstein, and M. Mead. The Case Research Strategy in Studies of Information Systems. *MIS Quarterly*, pages 369–386, 1987.

[5] J. Bloch. How to Design a Good API and Why it Matters. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, pages 506–507. ACM, 2006.

[6] J. Bloch. *Effective Java*. Pearson Education, India, 2008.

[7] N. Chapin, J. E. Hale, K. M. Khan, J. F. Ramil, and W.-G. Tan. Types of Software Evolution and Software Maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 13(1):3–30, 2001.

[8] S. Clarke. Measuring API Usability. *Doctor Dobbs Journal*, 29(5):S1–S5, 2004.

[9] J. Cohen. Weighted kappa: Nominal Scale Agreement Provision for Scaled Disagreement or Partial Credit. *Psychological bulletin*, 70(4):213, 1968.

[10] D. Dig and R. Johnson. The Role of Refactorings in API Evolution. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 389–398. IEEE, 2005.

[11] D. Dig and R. Johnson. How do APIs Evolve? a Story of Refactoring. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(2):83–107, 2006.

[12] Git. http://git-scm.com/. Accessed: 2015-05-08.

[13] J. Henkel and A. Diwan. Catchup!: Capturing and Replaying Refactorings to Support API Evolution. In *Proceedings of the 27th international conference on Software engineering*, pages 274–283. ACM, 2005.

[14] M. Henning. API Design Matters. *Queue*, 5(4):24–36, 2007.

[15] D. Hou and X. Yao. Exploring the Intent Behind API Evolution: A Case Study. In *Proceedings of the 2011 18th Working Conference on Reverse Engineering*, pages 131–140. IEEE, 2011.

[16] M. M. Lehman. On Understanding Laws, Evolution, and Conservation in the Large-program Life Cycle. *Journal of Systems and Software*, 1:213–221, 1980.

[17] M. M. Lehman. Programs, Life cycles, and Laws of Software Evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[18] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski. Metrics and Laws of Software Evolution - The Nineties View. In *Software Metrics Symposium, 1997. Proceedings., Fourth International*, pages 20–32. IEEE, 1997.

[19] B. P. Lientz and E. B. Swanson. *Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*. Addison-Wesley, 1980.

[20] T. McDonnell, B. Ray, and M. Kim. An Empirical Study of API Stability and Adoption in the Android Ecosystem. In *2013 IEEE International Conference on Software Maintenance*, pages 70–79. IEEE, 2013.

[21] D. G. Messerschmitt and C. Szyperski. Software Ecosystem: Understanding an Indispensable Technology and Industry. *MIT Press Books*, 1, 2005.

[22] M. Piccioni, C. A. Furia, and B. Meyer. An Empirical Study of API Usability. In *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*, pages 5–14. IEEE, 2013.

[23] R. Robbes, M. Lungu, and D. Röthlisberger. How do Developers React to API Deprecation? The Case of a Smalltalk Ecosystem. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 56. ACM, 2012.

[24] C. Robson. *Real World Research*, volume 2. Blackwell Publishers, Oxford, 2002.

[25] P. Runeson and M. Höst. Guidelines for Conducting and Reporting Case Study Research in Software Engineering. *Empirical Software Engineering*, 14(2):131–164, 2009.

[26] C. B. Seaman. Qualitative Methods in Empirical Studies of Software Engineering. *IEEE Transactions on Software Engineering*, 25(4):557–572, 1999.

[27] L. Shi, H. Zhong, T. Xie, and M. Li. An Empirical Study on Evolution of API Documentation. In *Fundamental Approaches to Software Engineering*, pages 416–431. Springer, 2011.

[28] J. Stylos, S. Clarke, and B. Myers. Comparing API Design Choices with Usability Studies: A Case Study and Future Directions. In *Proceedings of the 18th PPIG Workshop*, 2006.

[29] Z. Xing and E. Stroulia. Refactoring Practice: How it is and How it Should be Supported - An Eclipse Case Study. In *2013 IEEE International Conference on Software Maintenance*, pages 458–468. IEEE, 2006.

[30] R. K. Yin. *Case Study Research: Design and Methods*. Sage Publications, 2013.

The Impact of Developer Team Sizes on the Structural Attributes of Software

Ahmmad Youssef
Brunel University
London, United Kingdom
Ahmmad.Youssef@brunel.ac.uk

Andrea Capiluppi
Brunel University
London, United Kingdom
Andrea.Capiluppi@brunel.ac.uk

ABSTRACT

It is established that the internal quality of software is a key determinant of the total cost of ownership of that software. The objective of this research is to determine the impact that the development team's size has on the internal structural attributes of a codebase and, in doing so, we consider the impact that the team's size may have on the internal quality of the software that they produce.

In this paper we leverage the wealth of data available in the open-source domain by mining detailed data from 1000 projects in GoogleCode and, coupled with one of the most established of object-oriented metric suites, we isolate and identify the effect that the development team size has on internal structural attributes of the software produced.

We will find that some measures of functional decomposition are enhanced when we compare projects authored by fewer developers against those authored by a larger number of developers while measures of cohesion and complexity are degraded.

Categories and Subject Descriptors

D.1.5 [Programming Techniques]: Object-oriented Programming

General Terms

Measurement, Design, Economics, Human Factors.

Keywords

Open Source Software Development Process, Complexity Metrics

1. INTRODUCTION

To varying extents, the real and perceived success of an organisation depends on the quality - functional and non-functional - of the software it produces, commissions or purchases. There are many examples of organisations suffering significant loss resulting from poor software processes negatively affecting the quality of the software, eventually impacting an organisation's stakeholders through failed projects or serious software defects. The impact may lead to significant financial or reputational loss and can even be serious enough to cause an organisation to fail. At the very minimum, poor quality can prove a costly drain on resources. In his book 'The Economics of Software Quality' Capers Jones found that one half of most software development project budgets and two thirds of the typical development team's time are spent fixing poor quality [3]. Poor quality also leads to increased cost-to-change which can handicap an organisation's competitive position with a reduced ability to react to an increasingly dynamic world.

Clearly there is a need for a continual drive to understand the factors that can have a material effect on software quality. Software metrics will continue to play an important role in this process as they embody an empirical approach to software engineering that, if used appropriately, can lead to a significant reduction in the implementation and maintenance costs of the final software product. There are three general classifications of software metrics – product metrics, process metrics, and project metrics [4]. Our interest lies in product metrics and, more specifically, internal structural metrics.

One key factor that is clearly within the sphere of influence of management is the size of the development teams. While there has been significant work measuring the impact of team sizes on software process metrics (namely productivity) and limited work measuring impact of team sizes on external software metrics (namely fault-proneness), there has been no research investigating the impact of team sizes on the structural attributes of a codebase. As we will discuss further, filling this gap in the research would enable us to leverage the vast body of research linking object-oriented structural metrics to some key characteristics that matter greatly to practitioners such as testability and maintainability. In doing so, we will pave the way to gaining a significantly greater understanding of the true impact of team sizes on internal software quality.

2. RELATED WORK

2.1 Team Sizes and External Software Metrics

There has been plenty of valuable in-depth research investigating the relationship between development team productivity and its size. In his popular book 'The Mythical Man Month', Brooks argues that, since software development is a complex task, the communication effort is great and adding more developers can lengthen rather than shorten the time taken to complete a task as it adds an exponentially greater number of necessary communication paths between developers [5].

Roger et al., using data from 130 projects, empirically tested the impact of a number of factors on software development productivity concluding that only team size significantly impacts software development time and productivity. This was since

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

IWPSE'15, August 30, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3816-5/15/08...$15.00
http://dx.doi.org/10.1145/2804360.2804365

independently confirmed using other empirical methods (other papers) [6].

Although the emphasis in the research community has largely been on establishing the link between team sizes and productivity, there has been some work linking team sizes to measures used as a surrogate for software quality. Nagappan leveraged data from Microsoft's Windows Vista project to establish that metrics based on organisational structures – of which team sizes were one aspect - are a significant predictor of software failure-proneness. [7]

2.2 Internal Software Metrics

The study and application of internal software metrics dates back to the mid-1960's when the primitive Lines of Code metric was routinely used as the basis for measuring software development productivity (LoC per month) and quality (defects per KLoC).

In 1971 Akiyama proposed the use of metrics for software quality prediction proposing a regression-based model for module defect density (number of defects per line of code) where line of code was used as a crude indicator of complexity [19]. This was one of the earliest attempts, albeit a simplistic one, to extract an objective measure of software quality through the analysis of observables of the system. With the increasing diversity of programming languages, it became necessary to introduce a more nuanced model of software complexity.

McCabe and Halstead made significant contributions but with the increasing adoption of Object-Oriented (OO) programming languages, Chidamber and Kemerer argued that this new development necessitated measures that could guide organizations to its successful adoption. This fact, coupled with criticisms of existing metrics suites, saw the development of the Chidamber and Kemerer (CK) metrics suite [19].

Academic efforts to extend, validate and refine complexity metrics have been a dominant feature of metrics research ever since. Basili et al., motivated by the desire to leverage software metrics to provide guidance to the areas of a system where testing efforts are best spent, built on Henry's research [18] to establish the utility of the Chidamber and Kemerer software metrics suite as a predicator of fault prone software classes. This was achieved through the diligent assembly of eight software development teams and a thorough regression analysis to establish relationships between OO metrics and observed defects [9].

These are, by no means, the only studies of this nature. Subramanyam et al. conducted similar work with access to a large number of in-house developed codebases and were able to control for programming language and software size, confirming the results obtained by Basili et al. – results which were further validated in a multitude of similar studies, each adding its own unique dimension, whether on the analysis side or the case study subject [10][11][12][13][14].

More recently Saberwal et al. employed similar regression models to correlate CK metrics with bad code smells driven by the desire to guide refactoring efforts to where they are most needed [15]. Badri et al., using similar techniques, concluded that a correlation exists between LCOM and unit test coverage, validating the use of OO metrics as a predictor of the testability of classes [16].

3. RESEARCH PROBLEM

The objective of this research is to establish the impact that a development team's size has on the structural attributes of a codebase. The main caveat when embarking on such a study is that we must remain conscious that codebases with a larger team size are more likely to exhibit higher complexity than codebases with fewer developers simply due to the fact that the larger team is likely to have gone through more iterations of development as more complex functionality is implemented. To elaborate, when using version control systems, it is usual to build functionality iteratively through repeated modification of source files. It has been proven that, as software projects evolve, iterations of a codebase tend to exhibit growing complexity [8]. A key part of our approach to solving our research problem is to isolate and remove any impact that this effect may have on the metrics of an evolving codebase and to observe the impact of the development team size alone. We detail our approach to this particular challenge in section 4.4.

4. METHODOLOGY

4.1 Data Set

Given the wealth of project data accessible in the open source space, this was decided to be the best source of raw data for analysis. There are a number of popular open source project hosts (or 'forges') – GITHub, SourceForge, and GoogleCode. In terms of developer activity, GITHub is the most popular, followed by SourceForge and then GoogleCode [1]. Each of these forges has a unique and varied make-up of languages constituting its project population. It was considered that Java projects would be the most preferable to study given the myriad of available static code analysis tools readily available as well as the large number of projects available for study. Furthermore, Java consistently rates as the Object Oriented language with the highest adoption rates [2], an important consideration when bearing in the mind the need to ensure the relevance of this research to practitioners.

Table 1 A summary of the CK metrics and guidelines

Metric	Full Name	Attributes Measured
CBO	Coupling Between Objects	Count of other objects to which the object being considered is coupled. A high number can indicate poor encapsulation, a low level of reusability, and create difficulties in modification or testing.
DIT	Depth of Inheritance tree	A measure of complexity as measured by the number of parent classes from which a class may inherit behavior. A high number can point towards excessive design complexity.
LCOM	Lack of Cohesion of Methods	Measurement of the disparateness of functionality within an object. A high number can point towards poorly designed objects that do not adhere to the "single responsibility principle".
NOC	Number Of Children	A measure of reuse and abstraction. A high number can point towards poor design and diluted abstraction.
RFC	Response For a Class	Count of methods which may be executed in response to a message. High numbers may highlight objects with undue complexity complicating build, maintenance, and testing.
WMC	Weighted Methods per Class	An indicator of the complexity of a class through the method count in that object. A high number can indicate undue complexity and limited scope for re-use.

GoogleCode was the forge selected for study both for its popularity and high level of Java adoption rates. One final benefit of specifically mining GoogleCode was that project administrators can choose from among three available version control systems – Subversion, GIT, and Mercurial. This ensured that our toolchain needed to be compliant with each version control system meaning that, as GoogleCode itself prepares for eventual shutdown, the toolchain is equally suited to be re-used to mine GitHub (which uses GIT) or SourceForge (which uses Subversion and Mercurial).

4.2 Metrics Suite

The popular metrics suite proposed by Chidamber and Kemerer [12] is both well understood and has a significant supporting body of research. The details of this metric suite are outlined in table 1.

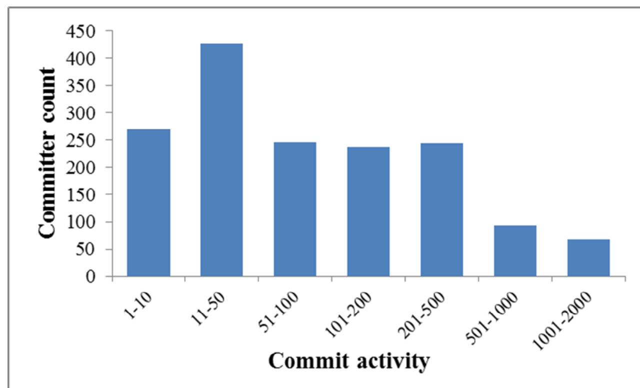
4.3 Defining the Development Team

There are a number of possible definitions of a software development team. Both Capra et al. and Smith et al. consider a team to consist of all developers to have worked on a codebase for any length of time [24][25] while Nagappan et al. (using data from IBSG [26]) consider the development team to also include management, administration and operations personnel.

Given the context of mining open source repositories, we favour defining the team size as the cumulative total of all unique committers present in the revision history in the version control system of a given project. Our view is that this definition is consistent with the prior art, simple to measure and reproduce, and elegantly allows us to capture the number of unique development design approaches that may have influenced the evolution of a codebase.

There are some potential limitations to this approach, most notably that we do not distinguish between frequent committers and causal (infrequent) committers. Figure 1 shows the relative activity levels of the committers in our data sample and, while it is true that the majority of commit activity takes place by a minority of committers, clearly the majority of committers do make a significant contribution and cannot be discounted.

Figure 1 The number of committers exhibiting a activity in a defined range against the number of commits in that range.



4.4 Data Analysis Techniques

In order to investigate the relationship between team sizes and internal quality metrics through mining open source repositories, there is a large amount of data that needs to be collated and analysed. This data essentially takes the form of a large population of file-level CK metrics along with meta-data associated with each file revision. This meta-data allows us to establish the number of unique committers to an individual project which is, to all intents and purposes, the project development team size.

Given the above data, we can group metrics together by project team size - irrespective of the individual project from which they came - and consider them distinct populations. For example, if project X and project Y each had *n* unique committers, all metrics belonging to each file within both projects would reside in a single bucket. Using this bucketing approach, we would have a limited number of distinct populations of metrics which could then be compared using statistical techniques.

As mentioned earlier, we must remain conscious that codebases written by larger teams are more likely to exhibit higher complexity than codebases with less developers given that a larger team will solve more complex problems and a codebase will typically go through more iterations (or 'revisions') in the process. This effect is illustrated in figure 2 where averaging metric values across files modified by a given number of committers yields a steady upward on measures of complexity.

Figure 2 Average metric value for files plotted against the cumulative number of committers to have edited the files.

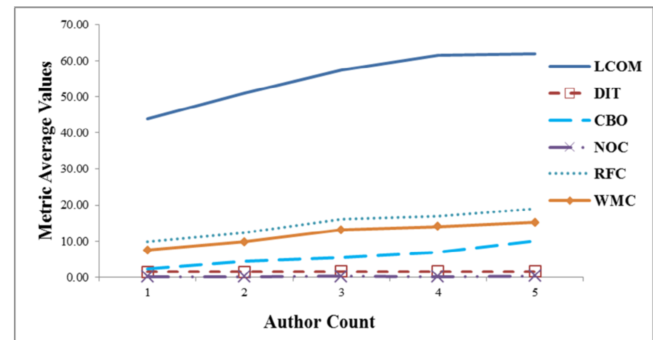
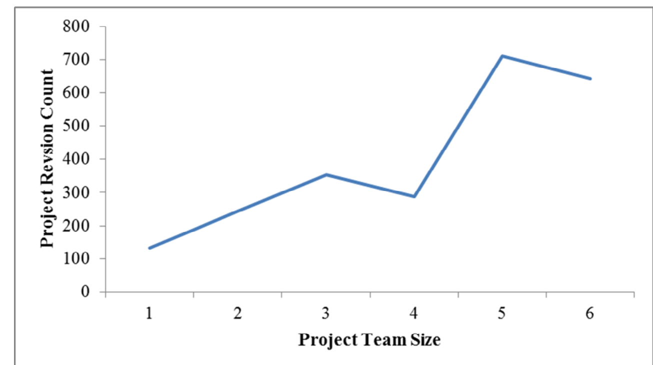


Figure 3 shows the increase in total project revision count against project team size. We can see this impact of this in Figure 4 where we average the metrics values and plot their progression by revision count.

Figure 3 Project revision count plotted against project team size



For this reason, our data analysis will take on an additional dimension alongside team size – namely that of the file revision count. From the meta-data associated with each revision, we can determine how many revisions any one file has undergone. This data will feed into our bucketing process where we can ensure that the population of metrics within a particular bucket only contains those metrics belonging to projects with a particular team size and only from files that have been modified a particular number of times. This approach, illustrated in figure 5, will give us confidence that when comparing our bucketed metric populations, any statistically significant differences are solely down to team size rather than fact that larger teams typically work on larger projects.

When comparing metrics populations bucketed by developer count, the Mann-Whitney test is ideally suited as all metrics populations are independent and consist of continuous data that we found not to be normally distributed.

Figure 4 Average metric values against revision count

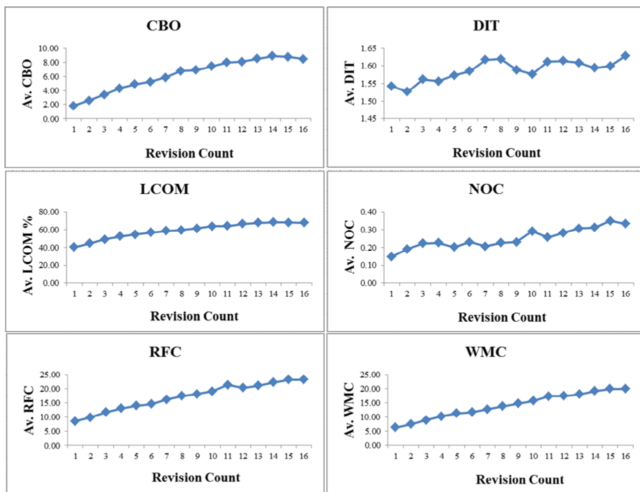
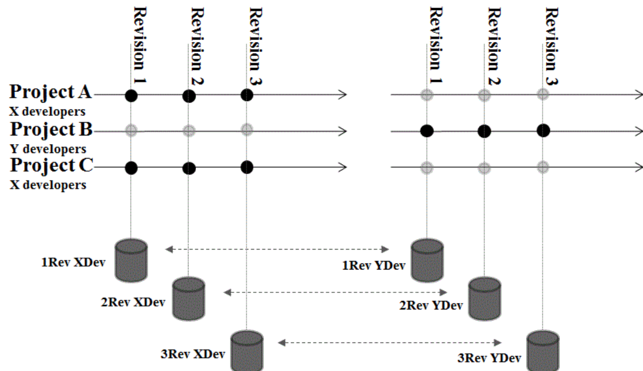


Figure 5 Bucketing approach illustrated



5. DATA COLLECTION

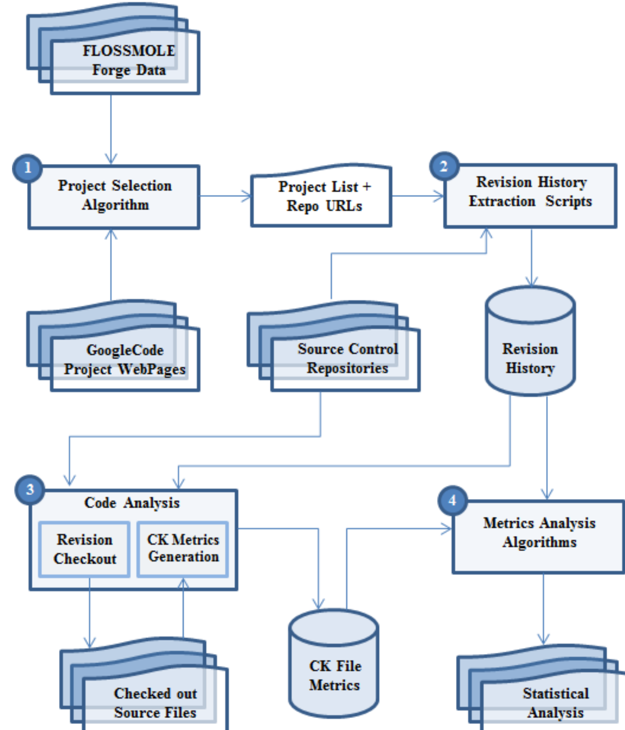
By leveraging the FLOSSmole project [20] we were able to obtain raw data describing, at a project level, details about all projects hosted in GoogleCode. This, along with the GoogleCode project webpages and the associated repositories formed the input into a bespoke toolchain illustrated in figure 6.

It is important to note that the majority of the complexity in this toolchain resides in the Project Selection Algorithm (1), the CK metrics Generation (3), and the Metrics Analysis Algorithms (4). Although the scripts to extract revision history (2) and checkout file revisions (3) may be considered a duplication of effort in prior research (most this work has notably been implemented in CVSAnalY [21]), it was felt that integration challenges in using the such a software package would outweigh the effort we would expend in developing our own bespoke scripts for the relatively simpler parts of the toolchain. For the more complex functionality of metrics generation, an off-the-shelf tool for metrics generation was employed. Our toolchain is illustrated in figure 6.

5.1 Project Selection Algorithm

This component (marked as component 1 in figure 6) is written in Java and takes in flat files made available by FLOSSmole detailing all the available projects hosted by GoogleCode and the ‘tags’ associated with each project. The project data is used to extract all projects with the tag ‘Java’ as of May 2012 – yielding us a list of 22594 GoogleCode hosted Java projects. That list was then reduced from 22594 projects to a more manageable subset by employing the pseudorandom ‘Math.random’ function in Java to select a number between 0-1 to be multiplied by the total number of projects until 1000 projects had been selected. In the process we discard from consideration any projects with no revision history as they represent projects which were not started and have no significance in this study and should not constitute part of our 1000 project sample.

Figure 6 Toolchain to extract and analyse revision based metrics



Once the projects were selected, the algorithm then extracts (via ‘screen-scraping’) the repository URL from the relevant project’s

page on the GoogleCode website. The project list with the associated URLs are consolidated in a single file which is used to drive the next part of the toolchain.

5.2 Revision History Extraction Scripts

The revision history extraction scripts (marked as component 2 in figure 6) are a relatively simple collection of shell scripts (that are runnable on a unix platform) to query, for all the projects in the input file, the relevant version control repositories to obtain the full revision history, storing it in a simple format to allow it to drive the code analysis stage.

5.3 Code Analysis Component

The code analysis component (marked at component 3 in figure 6) comprises, again, fairly simple shell scripts responsible for checking out each version of the project, handing over the heavy lifting of project metrics generation to run a metrics generation tool called 'Understand' by Scientific Toolworks Inc. [28]. The report created by Understand is of a particular format which is then passed through a file parser (written in Java) which extracts the information that is pertinent to our research and stores it in a format appropriate to our metrics analysis component. Understand version 2.6.610 was chosen as it is available on academic license and offers a unix-based command line tool that generates metric reports in an easily parsable format. The calculations used by Understand to generate metric values are in table 2.

Table 2 A description of how CK Metric values are calculated for Java classes by Understand

Metric	Full Name	Calculation for Java classes in 'Understand'
CBO	Coupling Between Objects	Number of other Classes invoked from this class. Library classes not included.
DIT	Depth of Inheritance tree	Number of parent classes in total
LCOM	Lack of Cohesion of Methods	For each member variable calculate the percentage of methods which do not access that variable. Average the percentages to determine LCOM.
NOC	Number Of Children	Count of other classes that directly extend it.
RFC	Response For a Class	Number of total methods including all methods in parent classes (regardless of invocation or visibility).
WMC	Weighted Methods per Class	Count of all methods in that class only (regardless of invocation, visibility, and instance or static).

5.4 Metrics Analysis Algorithms

The metrics analysis algorithms (marked as component 4 in figure 6) are one of the more complex components in our toolchain. It is a software package written in Java, and responsible for retrieving the generated metrics data, implementing the bucketing strategy, and running statistical tests to produce our analysis.

6. RESULTS

6.1 Bucket Populations

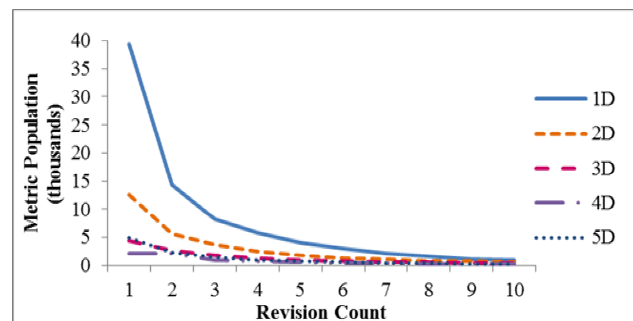
When we conduct a simple Mann-Whitney test for each CK metric type comparing two groups – the first being all metric results belonging to single developer projects and the second group being metrics results from all other projects - we find that the two groups are independent with p-values < 0.05. However, as discussed in detail in section 4.3, when analysing metrics results we bucket our data by revision count and project team size in order to isolate and observe the effect of team size alone.

It is logical that there would be a greater number of metric results pertaining to the lower revision counts as, by necessity, for a file to be revised, say, 5 times, it would have 4 prior revisions. However, it is perfectly normal for a file to only have fewer than 5 total revisions. This is an important consideration as buckets belonging to higher revisions and team sizes will have diminishing populations. Figure 7 and table 3 clearly show this effect. For our analysis, we only consider buckets belonging to team up to 5 developers strong with a maximum of 6 revisions as we see marked drop-off in bucket population sizes as team size and revision counts increase.

Table 3 Bucket population sizes

	PROJECT TEAM SIZE				
Rev	**1 Dev**	**2 Dev**	**3 Dev**	**4 Dev**	**5 Dev**
1R	39460	12524	4372	2060	4962
2R	14236	5552	2630	2169	2316
3R	8291	3662	1758	981	1509
4R	5743	2395	1282	695	973
5R	4024	1705	995	515	687
6R	2985	1311	786	393	528

Figure 7 Average metric values against revision count



6.2 Comparing Lone Developer Projects against Multi-developer Projects

Table 4 details the results from the statistical tests run across each team-size comparison. The first part of the table summarises the comparison results for metrics from single

developer projects against projects authored by two developers. Taking the WMC column as an example, we can see that the 2R bucket (this refers to the bucket of WMC metric values belonging to the second revision of files only) has statistically significantly lower values than in the single developer bucket. To be clear, this means that a Mann-Whitney test yields a p-value of <0.05 and we can observe higher median values in the single developer bucket. This means that we can say with a confidence level of 95% ±5% that files with two revisions and a single developer have higher WMC values than files with two revisions and two developers.
The next bucket down - 1D3R v 2D3R - exhibits no significant difference between the metrics populations for WMC.

In table 4 we can clearly observe that a large number of buckets that show statistical significance across CBO, DIT, LCOM and, to a lesser extent RFC and WMC. In the case of DIT, we can see a progression where more buckets show significance as the team size grows. In the case of other metrics we can see an inconsistent pattern with more buckets showing differences in the 1D v 3D and 1D v 4D tests. Another key observation is that CBO, DIT, RFC and WMC generally trend downwards as project team sizes increase while LCOM increases.

Table 4 Results of Mann-Whitney tests comparing the lone developer bucket of the various revision counts against the corresponding buckets for 2, 3, 4 and 5 developer buckets. The percentages relate to the proportion of the six buckets (1R-6R) that show p-values<0.05 for a particular team size comparison.

1D v 2D	CBO	DIT	LCOM	NOC	RFC	WMC
	17%	0%	0%	0%	0%	17%
	1R -	1R -	1R -	1R -	1R -	1R -
	2R -	2R -	2R -	2R -	2R -	2R >
	3R >	3R -	3R -	3R -	3R -	3R -
	4R -	4R -	4R -	4R -	4R -	4R -
	5R -	5R -	5R -	5R -	5R -	5R -
	6R -	6R -	6R -	6R -	6R -	6R -

1D v 3D	CBO	DIT	LCOM	NOC	RFC	WMC
	83%	17%	67%	0%	33%	33%
	1R -	1R -	1R >	1R -	1R >	1R >
	2R >	2R <	2R -	2R -	2R >	2R >
	3R >	3R -	3R <	3R -	3R -	3R -
	4R >	4R -	4R <	4R -	4R -	4R -
	5R >	5R -	5R -	5R -	5R -	5R -
	6R >	6R -	6R <	6R -	6R -	6R -

1D v 4D	CBO	DIT	LCOM	NOC	RFC	WMC
	83%	67%	67%	0%	33%	17%
	1R -	1R >	1R -	1R -	1R >	1R -
	2R <	2R <	2R -	2R -	2R <	2R >
	3R >	3R -	3R <	3R -	3R -	3R -
	4R >	4R -	4R <	4R -	4R -	4R -
	5R >	5R <	5R <	5R -	5R -	5R -
	6R >	6R <	6R <	6R -	6R -	6R -

1D v 5D	CBO	DIT	LCOM	NOC	RFC	WMC
	17%	83%	0%	0%	33%	50%
	1R <	1R -	1R -	1R -	1R -	1R -
	2R -	2R <	2R -	2R -	2R -	2R -
	3R -	3R <	3R -	3R -	3R -	3R -
	4R -	4R <	4R -	4R -	4R -	4R >
	5R -	5R <	5R -	5R -	5R >	5R >
	6R -	6R <	6R -	6R -	6R >	6R >

6.3 Increasing the Developer Count

Table 5 displays the result of a series of comparisons between projects with two developers against projects with 3, 4, and 5 developers respectively. We see very similar trends to the previous set of results in section 6.2.

7. CONCLUSIONS AND FUTURE WORK

Table 6 presents a summary of the discernable trends across each metric type as we add developers to a project team. We can see very similar trends regardless of whether we compare projects with a single developer against those with several developers, or whether we compare projects with multiple developers with those with still more developers.

Table 5 Results from the 5 developer against 2, 3, and 4 developer comparisons.

2D v 5D	CBO	DIT	LCOM	NOC	RFC	WMC
	17%	17%	0%	0%	17%	0%
	1R <	1R -	1R -	1R -	1R -	1R -
	2R -	2R -	2R -	2R -	2R -	2R -
	3R -	3R -	3R -	3R -	3R -	3R -
	4R -	4R <	4R -	4R -	4R -	4R -
	5R -	5R -	5R -	5R -	5R -	5R -
	6R -	6R -	6R -	6R -	6R >	6R -

3D v 5D	CBO	DIT	LCOM	NOC	RFC	WMC
	67%	50%	33%	0%	33%	50%
	1R <	1R -	1R -	1R -	1R <	1R <
	2R <	2R -	2R -	2R -	2R <	2R <
	3R -	3R <	3R >	3R -	3R -	3R -
	4R <	4R <	4R >	4R -	4R -	4R -
	5R -	5R <	5R -	5R -	5R -	5R -
	6R <	6R -	6R -	6R -	6R -	6R >

4D v 5D	CBO	DIT	LCOM	NOC	RFC	WMC
	67%	17%	67%	0%	50%	67%
	1R <	1R -	1R -	1R -	1R <	1R -
	2R >	2R -	2R -	2R -	2R >	2R -
	3R -	3R -	3R >	3R -	3R -	3R >
	4R <	4R <	4R >	4R -	4R -	4R >
	5R -	5R -	5R >	5R -	5R -	5R >
	6R <	6R -	6R >	6R -	6R >	6R >

Table 6. Summary of observed trends. Shaded boxes indicate a negative impact. Clear boxes indicate a positive impact.

Metrics	Objective	Lone developer v. Multi-developer teams	Smaller teams v. larger teams
CBO	↓	↓	↑
DIT	↓	↑	↑
LCOM	↓	↑	↑
NOC	↓	-	-
RFC	↓	↓	↓
WMC	↓	↓	↓

The clear conclusion is that where projects are collaborated on by a larger number of developers we are likely to see a decrease in cohesion (reflected by larger LCOM values), an increase in

structural complexity (reflected by larger DIT values) and an increase in coupling (reflected by larger CBO values). On a more positive note, we are likely to see that classes have improved functional decomposition (reflected by lower RFC and WMC values).

Finally, we can see consistency between our results and the research of both the work of Nagappan et al. [7] who linked larger team sizes with increased fault-proneness and Basili et al. confirmed that higher values of CBO and DIT, as observed in our research, is highly correlated with increased defect counts [9].

The implications of these findings are of relevance to practitioners. We believe that software development teams should take note of the fact that the structural attributes of a codebase can show degradation in some aspects as team sizes grow and the diligent use of tools like SonarQube [27] can provide visibility of this to developers and management in order for them to work together to mitigate any negative trends. This work is also relevant to the research community, to whom we suggest a number of avenues that we encourage the research community to pursue.

Firstly, we can hypothesize that the reasons driving the trends observed in table 6 are that, while a competent developer with relative unfamiliarity with the codebase will be capable of implementing code with a high degree of functional decomposition (for example, smaller single purpose methods), achieving low coupling and complexity as a codebase evolves typically requires a more in-depth understanding of the entirety of the codebase – an understanding that we can reasonably hypothesize is more likely to be lacking in a larger development team. Through qualitative analysis and engaging development team members, the research community could shed more light as to the drivers behind the trends revealed in this study.

Secondly, we believe that there is value in looking at how this work applies within the context of Agile development [22]. For example the Agile methodology recommends that development teams are co-located to facilitate communication between members [23]. In contrast, open source project teams tend to consist of people collaborating without necessarily sharing the same physical space. There is value in understanding if the negative effects of a larger development team manifest when a team is co-located and experiences lower barriers to effective communication.

Finally, we appreciate that a team with a total of 5 developers cannot necessarily be classed a large team – indeed this is considered the lower limit of the ideal Agile development team. We would be interested to see how these trends continue when taken up to and beyond the maximum ideal team size of 9 as stipulated by the Agile approach [23].

8. REFERENCES

[1] S. O'Grady, What Black Duck Can Tell Us About GitHub, Language Fragmentation and More, RedMonk, 2011, www.redmonk.com/sogrady/2011/06/02/blackduck-webinar (accessed 15/05/2015).

[2] Tiobe Software, TIOBE programming community index for June 2013, 2013, www.tiobe.com (accessed 15/05/2015)

[3] C. Jones, O. Bonsignour, The economics of software quality. Addison-Wesley Professional, 2011.

[4] S. Kan, Software Quality Metrics Overview. Metrics and Models in Software Quality Engineering, 2002, pp. 85-120.

[5] F. Brooks, The Mythical Man-Month. Addison-Wesley, 1975.

[6] J. Rodger, P. Pankaj, A. Nahouraii, Knowledge Management of Software Productivity and Development Time. Journal of Software Engineering and Applications, 4(11), 2011, pp. 609.

[7] N. Nachiappan, B. Murphy, V. Basili, The Influence of Organizational Structure on Software Quality: An Empirical Case Study. Proceedings of the 30th international conference on Software engineering, 2008.

[8] R. Prather, An Axiomatic Theory of Software Complexity Measure. The Computer Journal, 27(4), 1984, pp. 340-347.

[9] V. Basili, R., & L. Briand, W. Melo, A Validation of Object-Oriented Design Metrics as Quality Indicators. IEEE Transactions on Software Engineering, 22(10), 1996, pp. 751-761.

[10] R. Subramanyam, M. Krishnan, Empirical Analysis of CK Metrics For Object-Oriented Design Complexity: Implications For Software Defects. IEEE Transactions on Software Engineering, 29(4), 2003, pp 297-310.

[11] K. El Emam, W. Melo, J. Machado, The Prediction of Faulty Classes Using Object-Oriented Design Metrics. Journal of Systems and Software, 56(1), 2001, pp 63-75.

[12] M. Tang, M. Kao, M. Chen, An Empirical Study on Object-Oriented Metrics. Proceedings of the Sixth International Software Metrics Symposium, 1999, pp. 242-249.

[13] J. Xu, D. Ho, L. Capretz, An Empirical Validation of Object-Oriented Design Metrics For Fault Prediction. Journal of Computer Science, 4(7), 2008, pp 571.

[14] R. Malhotra, A. Jain. Fault Prediction Using Statistical and Machine Learning Methods for Improving Software Quality. Journal of Information Processing Systems, 8(2), 2012, pp 241-262.

[15] H. Saberwal, S. Singh, S. Kaur. Empirical Analysis Of Open Source System For Predicting Smelly Classes. International Journal of Engineering Research & Technology, 2(3), 2013.

[16] L. Badri, M. Badri, F. Toure, An Empirical Analysis of Lack of Cohesion Metrics for Predicting Testability of Classes. International Journal of Software Engineering and its Applications, 5(2), 2011, pp. 69-85.

[17] S. Chidamber, C. Kemerer, A Metrics Suite for Object Oriented Design. IEEE Transactions on Software Engineering, 20(6), 1994, pp. 476-493.

[18] W. Li, S. Henry, Object-Oriented Metrics That Predict Maintainability. Journal of Systems and Software, 23(2), 1993, pp. 111-122.

[19] F. Akiyama, An Example of Software System Debugging. IFIP Congress 71(1), 1971.

[20] J. Howison, M. Conklin, K. Crowston, FLOSSmole: A Collaborative Repository for FLOSS Research Data and Analyses. International Journal of Information Technology and Web Engineering, 1(3), 2006, pp. 17–26.

[21] G. Robles, S. Koch, J, González-Barahona, J. Carlos, Remote Analysis and Measurement of Libre Software Systems by Means of the CVSAnalY tool. Proceedings of the 2nd ICSE Workshop on Remote Analysis and Measurement of Software System, 2004, pp. 51-55.

[22] L. Lindstrom, R. Jeffries, Extreme Programming and Agile Software Development Methodologies. Information Systems Management, 2005, 21(13).
[23] K. Schwaber, J. Sutherland, The Scrum Guide. Scrum.org, 2014, www.scrumguides.org/docs/scrumguide/v1/scrum-guide-us.pdf (accessed 15/05/2015).
[24] R. Smith, J. Hale, A. Parrish, An Empirical Study Using Task Assignment Patterns to Improve the Accuracy of Software Effort Estimation. IEEE Transactions on Software Engineering, 27(3), 2001, pp. 264-271.
[25] E. Capra, A. Wasserman, A Framework for Evaluating Managerial Styles in Open Source Projects. Open Source Development, Communities and Quality, 2008, pp. 1-14.
[26] The International Software Benchmarking Standards, www.isbsg.org (accessed 15/05/2015).
[27] SonarQube, www.sonarqube.org (accessed 15/05/2015).
[28] SciTools, www.scitools.com (accessed 15/05/2015).

Revisiting the Applicability of the Pareto Principle to Core Development Teams in Open Source Software Projects

Kazuhiro Yamashita
Kyushu University, Japan
yamashita@posl.ait.kyushu-u.ac.jp

Shane McIntosh
McGill University, Canada
shanemcintosh@acm.org

Yasutaka Kamei
Kyushu University, Japan
kamei@ait.kyushu-u.ac.jp

Ahmed E. Hassan
Queen's University, Canada
ahmed@cs.queensu.ca

Naoyasu Ubayashi
Kyushu University, Japan
kamei@ait.kyushu-u.ac.jp

ABSTRACT

It is often observed that the majority of the development work of an Open Source Software (OSS) project is contributed by a core team, i.e., a small subset of the pool of active developers. In fact, recent work has found that core development teams follow the Pareto principle — roughly 80% of the code contributions are produced by 20% of the active developers. However, those findings are based on samples of between one and nine studied systems. In this paper, we revisit prior studies about core developers using 2,496 projects hosted on GitHub. We find that even when we vary the heuristic for detecting core developers, and when we control for system size, team size, and project age: (1) the Pareto principle does not seem to apply for 40%-87% of GitHub projects; and (2) more than 88% of GitHub projects have fewer than 16 core developers. Moreover, we find that when we control for the quantity of contributions, bug fixing accounts for a similar proportion of the contributions of both core (18%-20%) and non-core developers (21%-22%). Our findings suggest that the Pareto principle is not compatible with the core teams of many GitHub projects. In fact, several of the studied GitHub projects are susceptible to the "bus factor," where the impact of a core developer leaving would be quite harmful.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*Team size*

General Terms

Measurement

Keywords

Core development team, Pareto principle, Open source

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

IWPSE'15, August 30, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3816-5/15/08...$15.00
http://dx.doi.org/10.1145/2804360.2804366

1. INTRODUCTION

Understanding open source software (OSS) communities, i.e., the groups that are responsible for developing and maintaining an OSS system, is as important as understanding OSS systems themselves. By studying OSS communities, we accumulate knowledge about how these communities manage highly distributed development teams [18, 21]. Such knowledge enables the OSS development model to augment or replace development models in proprietary settings.

At the heart of OSS communities are *core developers*, i.e., the developers who take a leading role in the development and maintenance of a software project. For instance, Nakakoji *et al.* [19] state that core developers are *responsible for guiding and coordinating the development of an OSS project. Core Members are those people who have been involved with the project for a relative long time and have made significant contributions to the development and evolution of the system.* Mockus *et al.* [18] define core developers as the most productive developers who have made roughly 80% of the total contributions. Although these heuristics slightly differ, researchers agree that the impact that core developers have on a project is large.

Recent studies have shown that a small number of developers make a large proportion of the code contributions [5, 6, 18]. Moreover, it has been shown that the number of core developers follows the *Pareto principle* (a.k.a., the 80-20 rule), i.e., 80% of the contributions are produced by roughly 20% of the contributors [8, 17, 24].

Although the prior work makes important strides towards understanding core teams in OSS, the conclusions are drawn based on a small sample size (i.e., 1-9 studied systems). Therefore, in this paper, we set out to revisit how the Pareto principle applies to core teams in a large sample of 2,496 GitHub projects. We study GitHub projects because GitHub is one of the most popular social coding platforms, and many successful OSS systems are developed on GitHub (e.g., `Rails`[1]). Through analysis of the 2,496 GitHub projects, we address the following two research questions:

(RQ1) *Does the proportion of core developers of GitHub projects follow the Pareto principle?*
While the actual proportion of core developers varies depending on the heuristic of core developers that we

[1] `https://github.com/rails/rails`

Table 1: An overview of the results of prior work.

Paper	Dataset	Result
Mockus *et al.* [18]	Apache and Mozilla	10 to 15 developers performed 80% of the contributions.
Dinh-Trong and Bieman [5]	FreeBSD	28 to 42 developers performed 80% of the contributions.
Koch and Shneider [17]	GNOME	52 developers (out of 301 developers) performed 80% of the contributions.
Goeminne and Mens [8]	Brasero, Evince and Wine	20% of developers performed 85%, 80% and more than 90% of the contributions in each project.
Robles *et al.* [24]	GNOME	The core group has been identified as the 20% most contributing committers.
Geldenhuys [6]	9 OSS projects	3%-9% of developers performed 80% of the contributions.

use, 26%-58% of projects have core teams that are too small ($\leq$ 10% of active contributors) or 5%-28% have core teams that are too large ($\geq$ 30% of active contributors) to be considered compliant with the Pareto principle.

(RQ2) *Is there any difference between the contributions of core and non-core developers?*
Surprisingly, we find that the proportions of core and non-core developer activity are very similar when we normalize them by their contribution rates. For example, bug fixing activity accounts for 18%-20% of core developer contributions and 21%-22% of non-core developer contributions.

The main contributions of this paper are:

- A large-scale analysis of the core teams of 2,496 GitHub projects.
- A comparative analysis of three heuristics for identifying core developers.

Paper organization. The remainder of the paper is organized as follows. Section 2 surveys related work. Section 3 describes our heuristics for identifying core developers in more detail. Section 4 provides an overview of the studied GitHub projects. Section 5 describes the design and results of our case study. Section 6 discusses findings from our study. Section 7 discloses the threats to the validity of our study. Finally, Section 8 draws conclusions.

2. RELATED WORK

2.1 Proportion of Core Developers

Prior work has also analyzed the proportion of core developers in OSS projects. Table 1 provides an overview of the results of the prior work and the datasets that were analyzed. Mockus *et al.* [18] hypothesize that the open source development model would rely on a team of core developers who control the code base and that these core developers would create 80% or more of the new functionality. Furthermore, Mockus *et al.* argue that the core team would be no large than 10 to 15 people based on analysis of the Apache and Mozilla projects. Crowston *et al.* [3] compared three approaches to identify the core developers within 116 SourceForge projects using bug fixing activity. Although the results differ among the three studied approaches, all of the approaches indicate that the core developers make up a small fraction of the total number of contributors. Goeminne and Mens [8] found evidence for the Pareto principle in three activities (development, email discussion and bug tracker activity) in three OSS projects. Robles *et al.* [24] arrived at similar conclusions — the core team makes up roughly 20% of the contributing committers.

On the other hand, other studies arrive at contradictory conclusions. For example, Dinh-Trong and Bieman [5] replicated Mockus *et al.*'s work using data from the FreeBSD project, finding that 28-42 out of 161-265 developers perform 80% of the contributions. Koch and Shneider [17] find that 52 out of 301 developers make 80% of the contributions in the GNOME project. Through analysis of nine systems, Geldenhuys [6] find that the proportion of core developers does not comply with the Pareto principle.

Much of the prior work analyzes a small number of subject systems. Hence, we set out to analyze core teams in a large number of systems. More specifically, we formulate the following research question:

> *(RQ1) Does the proportion of core developers of GitHub projects follow the Pareto principle?*

In addition to the size of core teams, Mockus *et al.* [18] hypothesized that a group, which is larger by an order of magnitude than the core team, will repair defects. From this hypothesis, we derive that non-core developers focus more on maintenance activity (e.g., bug fixing) than implementation activity. Goeminne and Bieman [8] showed that 2-6 out of the top 20 developers also contributed to plenty of the bug report and email discussions. However, to the best of our knowledge, the contribution activity of core and non-core developers have not been quantitatively compared. Hence, we formulate the following research question:

> *(RQ2) Is there any difference between the contribution activity of core and non-core developers?*

2.2 Studies on GitHub

In recent years, GitHub has become a popular source of data for SE researches. Gousios *et al.* [10, 11] focus on the pull-based development process. They first answer basic questions about what the life cycle of a pull request is, and how prevalent the pull-based development process is [10]. In more recent work, Gousios *et al.* also study the impact that the pull-based development process has on integrators, who manage code contributions [11].

Dabbish *et al.* [4] conducted an interview with GitHub users to find out what inferences people make from GitHub transparency, and what the value of transparency for soft-

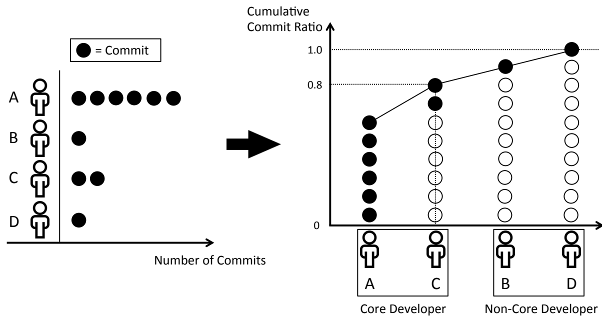


Figure 1: Identifying core developers using an example project.

ware development is. Their work reveals that four types of transparency-related and social inferences feed into three types of collaborative activities, such as project management. In the project management activities, the visibility of GitHub public repositories shifts development processes from being repository-focused to being owner-focused and contributor-focused.

Previous studies focused on the GitHub features. On the other hand, we focus on the proportion of core developers in GitHub projects.

3. HEURISTICS TO IDENTIFY CORE DEVELOPERS

In order to perform our study, we need to define heuristics to identify core developers. Inspired by previous studies, this paper explores three heuristics to identify core developer from the perspective of contributions as described below.

3.1 Commit-Based Heuristic

Several previous papers that have studied core developers [8, 18, 24] use the heuristic that defines core developers as those who produce roughly 80% of the total contributions. In this paper, we adopt this heuristic — after sorting the developers by their number of contributions in descending order, the core developers are those who have produced 80% of the project contributions, cumulatively.

For instance, Figure 1 shows an example project with four developers: A, B, C, and D. In order to determine core developers, we first sort the developers by the number of commits in descending order (A: 6, C: 2, B: 1 and D: 1). Next, we calculate the percentage of total commits that each developer has produced (A: 60%, C: 20%, B: 10%, and D: 10%). Then, we calculate the cumulative percentage (A: 60%, C: 80%, B: 90%, and D: 100%). Finally, we select core developers, one at a time, moving left to right, until we reach a cumulative percentage of 80%. In this example, A and C are identified as core developers.

In our algorithm, we do not handle the special case where there are some developers who have same number of commits on the border of core and non-core developers. We do not suspect that who we select to be a member of the core team should have a significant impact on the results, since: (a) these developers have produced the same number of contributions and (b) they are at the tail end of the core team contributions.

Table 2: Finding self-identified mirror projects.

Category	Used regular expression [16]	#Projects
Mirror Of	$mirror\ of\ .*repo\|git\ repo\ of$	10
Sourceforge	$sourceforge\|sf\backslash.net$	6
Bitbucket	$bitbucket$	2
Subversion	$\backslash W(svn\|subversion)\backslash W$	4
Mercurial	$\backslash W(mercurial\|hg)\backslash W$	0
CVS	$\backslash Wcvs\backslash W$	0
Total	-	23

3.2 LOC-Based Heuristic

Similar to the commit-based heuristic, the LOC-based heuristic defines core developers according to the size of the contributions that they make [17, 18]. While we conduct our experiments using three size metrics, i.e., *the number of added lines*, *the number of deleted lines* and *the churn* (the sum of the number of added and deleted lines), we find that the results are similar across the three metrics. Therefore, to conserve space, we show only the results for churn in the remainder of the paper. Similar to commit-based heuristic, we identify core developers as those who cumulatively contribute 80% of the churn.

3.3 Access-Based Heuristic

Core developers can also be defined as those who have be given direct write access to the main VCS repository. For example, in projects like PostgreSQL, only core members can record changes directly in the main VCS repository — other contributors must convince core developers to record their changes on their behalf [7]. Hence, we can also identify core developers from a VCS access perspective.

In GitHub, project owners can grant write access to the project's main repository to other contributors. GHTorrent collects this information using the *collaborators* API and stores it in the *project_members* column [9]. According to the description of the collaborators API, the list includes all organization owners and users with access rights.[2] Since this list may include users who are members of an organization, but who did not contribute to a project, we define the access-based core developers as those who appear in the access list and have also made at least one commit.

Unfortunately, we find that roughly half of the studied projects do not use the access-based feature of GitHub. These projects are filtered out of our analysis when we use the access-based heuristic.

4. DATASET

In this section, we describe how we prepare the dataset of GitHub projects for our study. Figure 2 provides an overview of our dataset preparation steps.

We begin our study with the collection of GitHub project data that is available via GHTorrent [9]. However, GitHub hosts a large number of repositories, many of which are not software projects. Hence, we filter the GHTorrent data according to the suggestions of Kalliamvakou *et al.* [16]. We take three steps to create our dataset from the available GitHub projects. Initially, GHTorrent includes 8,510,504 repositories.

[2] `https://developer.github.com/v3/repos/collaborators/`

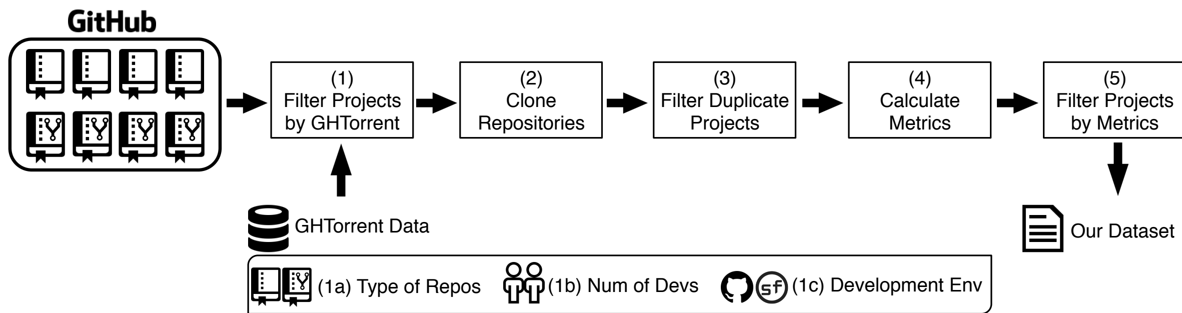


Figure 2: An overview of our data extraction approach.

(1) Filter Projects by GHTorrent Data

(1a) Type of Repository. In GitHub, there are two types of repositories: *main repositories* and *fork repositories*. A fork is a working copy of a main repository. Forking a repository allows developers to freely experiment with changes without interfering with the ongoing development of the original project.[3] In GitHub, fork repositories can contribute changes back upstream to the main repositories that they are forked from by issuing pull requests. If the maintainers of the upstream repository agree with the changes that are proposed by a pull request, the request is accepted, and the changes are integrated into the main repository. As all accepted pull requests are stored in main repository, we only extract commits from the main repository, ignoring commits that only appear in forks.

(1b) Number of Developers. Two types of authorship data are recorded in Git repositories. The committer is the team member who recorded the changes in the repository using the `git commit` command. The author is the team member who produced the code change itself. In this study, we focus on the authors of the changes, ignoring the committer data, since the author is the team member who actually produced the changes, while the committer is the team member responsible for the integration work.

Furthermore, since projects with a small number of developers can easily achieve extreme core team proportions, we filter away projects that have too few developers (number of developers < 10).

(1c) Development Environments. In this study, we would like to investigate core developers especially in projects that are developed on GitHub. Kalliamvakou *et al.* [16] find that GitHub is not only a popular social coding platform, it also serves as a popular host for mirrored repositories.[4] Since such mirrored projects may not be developed in the same manner as projects on GitHub, we need to filter them out of our dataset. To do so, we heed the advice of Kalliamvakou *et al.* [16]:

1. Avoid projects that have a large number of committers who are not registered GitHub users.
2. Avoid projects that explicitly state that they are mirrors in their description.

To address item 1), we filter away projects where less than 90% of the committers are registered GitHub users. To address item 2), we filter away projects with descriptions that match the regular expressions listed in Table 2, as proposed by the prior work [16].

After applying the filters of steps (1a)-(1c), 4,618 projects remain in our dataset.

(2) Clone Projects

Now that the number of projects has become manageable, we clone the selected repositories into our local environment to calculate the metrics that we use for our case study. Unfortunately, some of the projects that we select from the GHTorrent dataset are no longer available to be cloned (e.g., deleted repositories). Thus, we cannot include such projects in our dataset. Nonetheless, we could clone 4,154 projects.

(3) Filter Duplicated Projects

Even after handling explicitly forked repositories, there are still some duplicate repositories hosted on GitHub (i.e., cloned and registered repositories that were not created using the GitHub fork feature). Such projects do not count as fork projects, but those projects have largely the same history as their originals. Since such projects will introduce noise in our dataset, we first detect them using the steps below, and then filter them out of our dataset.

We use the hashes of commits (SHAs) recorded in the Git repositories to identify duplicated projects. We consider any repositories that shares more than 70% of the same commit SHAs as a copied repository. We remove both repositories from our dataset because it is often difficult to determine which repository is the original one and which one is the copy.

After removing these repositories, 3,533 projects remain in our dataset.

(4) Calculate Metrics from Repositories

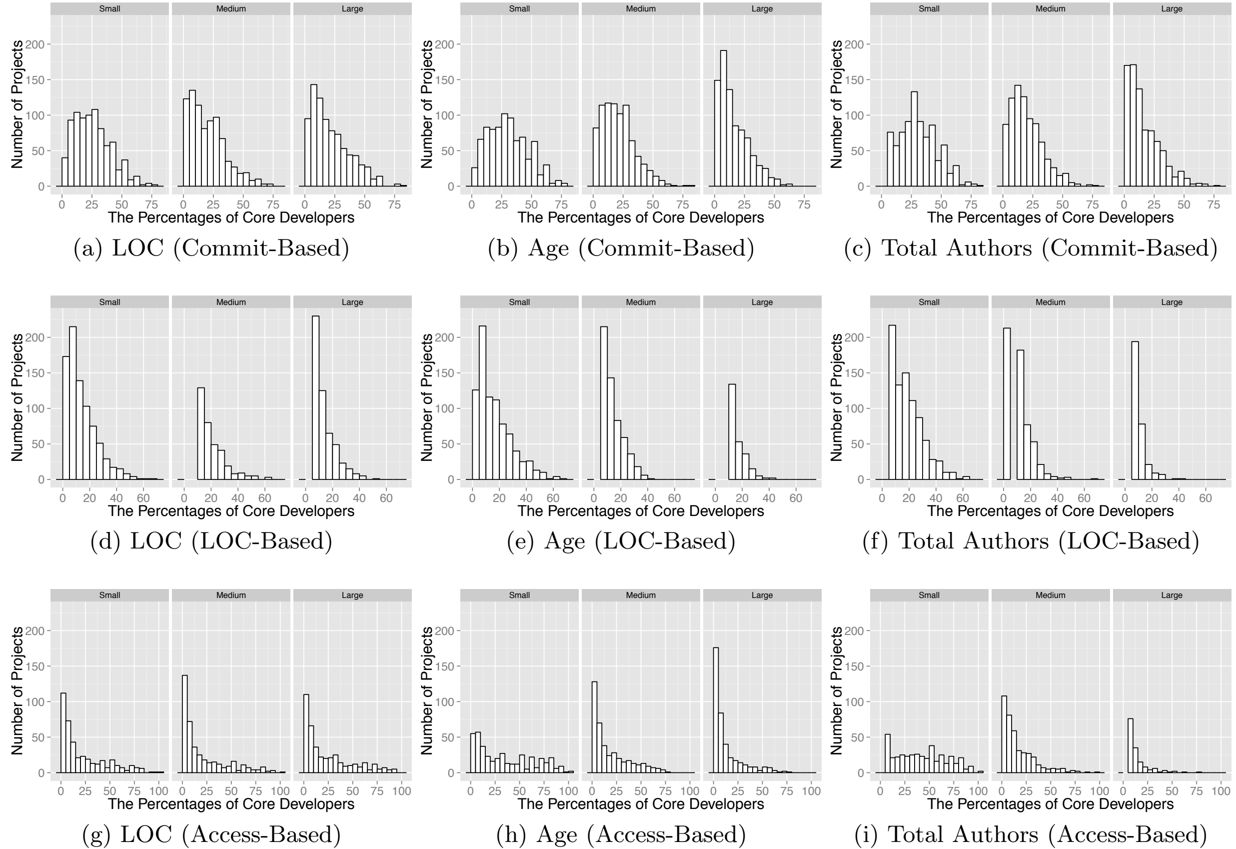
For the remaining projects, we calculate the metrics that are listed below in order to perform our case study.

LOC. We use *cloc*[5] to calculate LOC. Our LOC count does not include code comments or blank lines.

Total Commits. We count the number of commits by using the `git log` command with the `--no-merges` option.

Total Authors. We identify the unique authors by author name and email address, which we are able to extract from the commit logs. We use a tool[6] to disambiguate author

[3] `https://help.github.com/articles/fork-a-repo/`
[4] e.g., `https://github.com/apache`
[5] `http://cloc.sourceforge.net/`
[6] `https://github.com/bvasiles/ght_unmasking_aliases`



(a) LOC (Commit-Based) (b) Age (Commit-Based) (c) Total Authors (Commit-Based)

(d) LOC (LOC-Based) (e) Age (LOC-Based) (f) Total Authors (LOC-Based)

(g) LOC (Access-Based) (h) Age (Access-Based) (i) Total Authors (Access-Based)

Figure 3: Distribution of Projects of Each Size Categories

names and email addresses. We disambiguate names and email addresses of authors because some developers appear with slightly different forms [6].

Age. We calculate the age of a project (in days) by subtracting the time of the latest commit from the time of the initial commit.

(5) Filter Projects by Metrics

We not only filter projects that have fewer than 10 developers, but similar to Bissyande *et al.* [1], we also filter projects that have fewer than 1,000 LOC.

Finally, we obtain a dataset that includes 2,496 GitHub projects for the commit-based and LOC-based core developer heuristic. Since 1,284 of these projects do not have the information that is needed to detect contributors with write access (*cf.* Section 3), only the remaining 1,212 projects are studied using the access-based heuristic.

5. STUDY RESULTS

In this section, we present the results of our study with respect to our two research questions. For each research question, we present our approach and our results.

(RQ1) Does the proportion of core developers of GitHub projects follow the Pareto principle?

We begin our study by measuring the proportion of developers who are active enough to be considered core developers.

Approach. To address our first research question, we calculate the proportion of the development team that is considered to be part of the core team (*cf.* Section 3) of each studied project. Then, we use histograms to study the distribution of core team sizes in the studied projects.

The Pareto principle or the so-called "80-20 rule" states that 80% of the contributions are performed by roughly 20% of the contributors. In this study, similar to prior work [18, 23], we add a window of flexibility, considering projects where the core team proportion is $20\% \pm 10\%$ as being compliant with the Pareto principle. Indeed, Mockus *et al.* [18] showed that the core team proportions of modules in the Mozilla project are roughly 19%-25%. Moreover, Robles *et al.* showed that 10%-20% of developers produced more than 50% activities (in many cases as much as 90% or 95%).

We address RQ1 using two analyses. First, we analyze the distributions of proportions of core developers. Then, we split up the projects according to three confounding factors. Since the core team characteristics of smaller projects likely differs from those of larger projects, we divide the dataset into three strata (small, medium, and large) along three confounding factors (system size, team size and project age). We evenly divide the dataset accordingly, i.e., each stratum includes 832 projects for the commit-based and LOC-based heuristics, and each stratum includes 404 projects in the access-based heuristic. We then plot histograms of the core team proportions of projects in each of these nine strata. In this paper, we do not show the plots of overall distribution

Table 3: The spread of projects among strata of project size and age.

Heuristic	Size Metrics	Stratum	Proportion of Core Developers 0%-10%	10%-30%	30%-100%
Commit-Based	LOC	Small	143 (17%)	411 (49%)	278 (33%)
		Medium	264 (32%)	389 (47%)	179 (22%)
		Large	242 (29%)	368 (44%)	222 (27%)
	Age	Small	94 (11%)	359 (43%)	379 (46%)
		Medium	203 (24%)	447 (54%)	182 (22%)
		Large	352 (42%)	362 (44%)	118 (14%)
	Total Authors	Small	80 (10%)	365 (44%)	387 (47%)
		Medium	224 (27%)	449 (54%)	159 (19%)
		Large	345 (41%)	354 (43%)	133 (16%)
	General		649 (26%)	1,168 (47%)	679 (27%)
LOC-Based	LOC	Small	403 (48%)	367 (44%)	62 (7%)
		Medium	487 (59%)	304 (37%)	41 (5%)
		Large	557 (67%)	253 (30%)	22 (3%)
	Age	Small	354 (42%)	376 (45%)	102 (12%)
		Medium	501 (60%)	316 (38%)	15 (2%)
		Large	592 (71%)	232 (28%)	8 (1%)
	Total Authors	Small	227 (27%)	497 (60%)	108 (13%)
		Medium	502 (60%)	315 (38%)	15 (2%)
		Large	718 (86%)	112 (13%)	2 (0.2%)
	General		1,447 (58%)	924 (37%)	125 (5%)
Access-Based	LOC	Small	192 (48%)	100 (25%)	112 (28%)
		Medium	211 (52%)	92 (23%)	101 (25%)
		Large	177 (44%)	98 (24%)	129 (32%)
	Age	Small	115 (28%)	94 (23%)	195 (48%)
		Medium	202 (50%)	107 (26%)	95 (24%)
		Large	263 (65%)	89 (22%)	52 (13%)
	Total Authors	Small	60 (15%)	87 (22%)	257 (64%)
		Medium	191 (47%)	143 (35%)	70 (17%)
		Large	329 (81%)	60 (15%)	15 (4%)
	General		580 (48%)	290 (24%)	342 (28%)

Table 4: Distributions of projects according to the number of core developers.

Number of Core Developers	1-9	10-15	16-20	21-50	51-100	101-
Commit-Based	1,924 (77%)	273 (11%)	98 (4%)	137 (5%)	17 (0.7%)	47 (2%)
LOC-Based	2,397 (96%)	57 (2%)	15 (0.6%)	13 (0.5%)	4 (0.1%)	10 (0.5%)
Access-Based	1,036 (85%)	128 (11%)	24 (2%)	24 (2%)	0 (0%)	0 (0%)

to conserve space because we find that the distributions of the medium strata follow the same trends as the overall distributions.

Results. Figure 3 shows the core team distributions of the studied projects. Table 3 shows the exact numbers of projects of each category and percentile. In Table 3, the gray colored columns show the Pareto-compliant range.

Contrary to prior results, we find that the core team size of projects distributes broadly. Figure 3 and Table 3 show that the distributions are different according to the heuristic. Indeed, unlike prior work [8, 17, 24], we find that there are many projects that fall outside of our range of Pareto compliance (10%-30%).

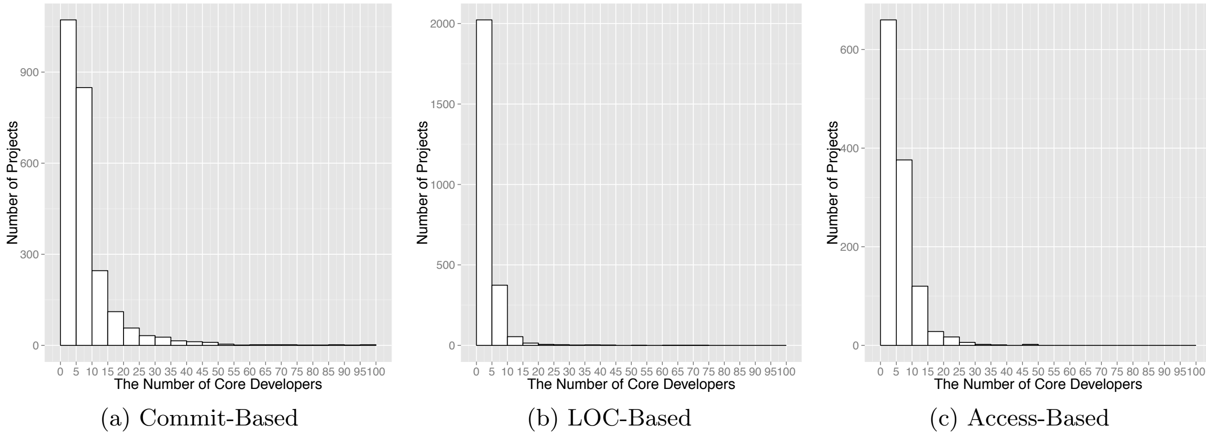
When we focus on each heuristic and confounding factor, we observe the following trends.

Commit-Based: Table 3 shows that, irrespective of the stratum, 43%-54% of the studied projects are Pareto compliant. When controlling for project age and team size, we find that the number of projects with the smallest core team size (i.e., 0%-10%) increases as we shift from the smallest to largest strata. On the other hand, this trend is not as extreme in the system size strata. Therefore, we conclude that project age and team size have a larger impact on the core team proportion than system size does.

LOC-Based: The LOC-based heuristic is more right skewed than the commit-based heuristic. Similar to the commit-based heuristic, Table 3 shows that the right skew increases as the system size increases. Moreover, the total number of authors seems to impact to the core team proportion because the difference between small and large stratum is the largest among the three studied metrics.

Access-Based: Figure 3 shows that the distributions of the access-based heuristic are similar to those of the LOC-based heuristic. However, there are more projects that fall in the 30%-100% range for the access-based heuristic than the LOC-based heuristic. Similar to the commit-based heuristic (Table 3), age and team size also appear to have an impact on the core team proportion of the access-based heuristic.

Figure 4 shows the number of core developers. In Figure 4, the x-axis shows the number of core developers and the y-axis



(a) Commit-Based (b) LOC-Based (c) Access-Based

Figure 4: The distribution of projects according to the number of core developers.

shows the number of projects. Table 4 shows the breakdown of projects stratified by the number of core developers.

From the perspective of core team size, we find support for the findings of prior studies [5, 17, 18]. Mockus *et al.* argue that if the core team uses only an informal means of coordinating, the group will be no larger than 10-15 people [18]. Conversely, Dinh-Trong and Bieman [5] find that 28-42 developers provide 80% of the contributions in the FreeBSD project. Koch and Schneider [17] find that 52 developers provide 80% of the contributions in GNOME project.

88%-98% of projects have fewer than 16 core developers. Unlike the proportion of core developers, the distributions of the number of core developers are similar across the studied heuristics. Indeed, Table 4 shows that 88%-98% of the studied GitHub projects have fewer than 16 core developers.

We further analyze the 2%-12% of projects that have more than 15 core developers to find out what kind of projects have larger core teams. When using the commit-based heuristic, 275 out of the 299 projects that have more than 15 core developers are categorized in large stratum of total authors and the remaining 24 projects are in medium stratum. When using the LOC-based heuristic, 41 of the 42 projects are in large stratum of total authors and the remaining one project is in medium stratum. When using the access-based heuristic, 27 of the 48 projects are in large stratum of total authors, and 20 of the remaining 21 projects are in medium stratum. These observations indicate that most of the projects that have many core developers also tend to a larger pool of contributors than the other projects.

Contrary to prior work, we find that there are several projects that have larger or smaller core team proportion than we consider to be compliant with the Pareto principle. Moreover, we find that most projects have 15 or fewer members of the core team.

(RQ2) Is there any difference between the contribution activity of core and non-core developers?

To address this RQ, we compare the types of contributions that are performed by core and non-core developers.

Table 5: Keywords used to classify commits [12].

Development	
Activity Type	Keywords
Forward Engineering	implement, add, request, new, test,start, includ, initial, introduc, creat, increas
Maintenance	
Activity Type	Keywords
Reengineering	optimiz, adjust, update, delet, remov, chang, refactor, replac, modif, (is, are) now, enhance, improv, design change, renam, eliminat, duplicat, restructur, simplif, obsolete, rearrang, miss, enhanc
Corrective Engineering	bug, fix, issue, error, correct, proper, deprecat, broke
Management	clean, license, merge, release, structure, integrat, copyright, documentation, manual, javadoc, comment, migrat, repository, code review, polish, upgrade, style, formatting, organiz, TODO

Approach. Previous studies have explored the purposes of changes [12, 14, 20]. In this study, we adopt Hattori and Lanza's approach to identify the purpose of changes. Hattori and Lanza [12] proposed a lightweight approach to classify each commit into development or maintenance activities based on the accompanying commit messages. They defined four main activities: *forward engineering* as development activity; and *reengineering*, *corrective engineering* and *management* as maintenance activity. They also provide keywords that are indicative of the type of activity (Table 5). *Forward engineering* activities implement new requests and add new features. *Reengineering* activities are related to refactoring, redesign and other actions to enhance the quality of the code. *Corrective engineering* activities fix defects. *Management* activities are other general maintenance activities that are not related to system functionality, such as code reformatting and documentation.

To ensure that the classification provided by Hattori and

Table 6: Developer Activity

Type of Activity	Commit-Based		LOC-Based		Access-Based	
	Core	Non-Core	Core	Non-Core	Core	Non-Core
Forward Engineering	15%	18%	16%	18%	17%	16%
Reengineering	29%	30%	29%	30%	24%	30%
Corrective Engineering	20%	21%	20%	22%	18%	21%
Management	14%	13%	14%	13%	12%	15%
Empty	0.1%	0.1%	0.1%	0.1%	0.3%	0.1%
Unknown	22%	17%	22%	17%	30%	19%
Total #of Commits	4,692,063	1,054,460	4,739,121	1,007,402	931,265	1,934,196

Lanza is sufficient for our dataset, we manually analyze a randomly selected sample of 384 commit comments. The sample is selected such that it provides a confidence level of 95% with a confidence interval of ±5%. The manual analysis reveals that some commits have an empty commit comment. We classify such commits as *empty*.

Hattori and Lanza's approach searches for keywords in commit messages in the following order: empty comments, management, reengineering, corrective engineering and forward engineering. The commit comments of so-called tangled changes [13] can match multiple purpose keyword types. For example, a developer can clean up code and fix a bug within one commit. For these commits, the approach classifies the commit according to the keyword that is found first (e.g., the commit described above is classified into reengineering activity). The commits that could not be classified into any of the classes are marked as *unknown*.

Using Hattori and Lanza's approach, we classify and compare the distributions of activities of core and non-core developers. In total, our commit-based and LOC-based heuristic datasets includes 5,746,523 commits, and our access-based one contains 2,865,461 commits.

Results. Table 6 shows the distribution of activities of core and non-core team members. Interestingly, the total number of commits that are contributed by core and non-core team members are very similar when we use either commit-based and LOC-based heuristics. On the other hand, the access-based heuristic shows that the number of commits of core developers is less than that of non-core developers. In this study, we only consider the authors of commits. Hence, this discrepancy between core and non-core contributions might show that many of the access-based core developers focus on integration work rather than writing code.

The proportions of contribution activity of core and non-core developers are similar. Irrespective of the core team heuristic, we find that the distributions of activities are very similar. Reengineering accounts for the largest proportion of activity for both core and non-core developers, with proportions ranging between 24%-30%. In the other type of activities, the difference between the proportion of activity of core and non-core developers is at most 6 percentage points. Therefore, we conclude that the difference in activity proportions between core and non-core is negligible.

The proportions of contribution activity of core and non-core developers are similar.

6. DISCUSSION

6.1 The Bus Factor

We find that more than half of the studied projects have a core team comprised of (at most) 20% of the pool of active developers and more than 88% of the studied projects have a core team of (at most) 15 developers. These results indicate that many projects have a low *bus factor* [2, 22, 25], i.e., face the risk of key personnel leaving the project. Ye and Kishida [26] find that development of GIMP was once halted because a key core developer left the project. To avoid such cases, projects must share knowledge among developers.

On the other hand, similar to the work of Dinh-Trong and Bieman [5], we find that there are projects that have large core teams. In this study, we just show the distribution and do not investigate each of the projects deeply. In future work, we plan to conduct a deeper analysis of projects with large core teams. For example, investigating whether or not such projects have well-defined mechanisms for developer promotion rather than the informal arrangements that Mockus *et al.* [18] hypothesized could yield fruitful results.

6.2 Core and Non-core Developer Activity

Prior work [18] hypothesized that a group larger by an order of magnitude than the core team will repair defects. If the hypothesis is true, we assumed that the proportion of maintenance activity of non-core developers is large. However, our results show that both types of developers have similar proportions of development activities. Furthermore, when we consider the number of corrective engineering commits, the number of the commits by core developers is much larger than that by non-core developers.

Our results may be a characteristic of the GitHub development environment. With the growth of social coding platforms (e.g., GitHub), the nature of core teams in modern OSS projects may have changed. For example, GitHub projects boast a higher rate of acceptance for contributions than the OSS projects of the past did. Indeed, while Jiang *et al.* [15] find that only 33% of contributions are eventually integrated into the Linux kernel (one of the largest OSS projects, which mainly developed by outside of GitHub), Gousios *et al.* [10] find that 84% of contributions are eventually integrated into GitHub projects.

6.3 The Impact of Thresholds

In this study, we filter projects to remove immature software projects by using some thresholds, i.e., the total authors and LOC (*cf.* Section 4). As such, our results may be sensitive to these thresholds. To check for threshold sensitivity, we re-apply our analysis using other threshold values (total

Table 7: **The proportion of projects that are Pareto compliant when we use other threshold values.**

Metric	Threshold	#ofProjects	Proportion
Total Authors	5	2,526	46%
	20	1,664	49%
LOC	500	2,685	46%
	2,000	2,220	47%

authors = 5, 20 and LOC = 500, 2,000) and discuss changes to our results below.

Table 7 shows the proportion of projects that are Pareto compliant when we vary the thresholds. Irrespective of the threshold, similar to our results in Section 5, we observe that more than half of projects are not Pareto compliant. These results suggest that while our results slightly vary when the thresholds change, the main conclusions are not heavily impacted.

7. THREATS TO VALIDITY

7.1 Construct Validity

In this paper, we adopt three heuristics to identify core developers. The commit-based and LOC-based heuristics are based on the amount of contribution to the product. Even though there are a lot of metrics that can capture contribution units, the amount of contribution is one of the most basic metrics that is used to identify core developers. Moreover, previous studies that focus on core contributors [5, 6, 8, 17, 18, 24] also conduct their analysis from the perspective of the amount of contribution. Therefore, we feel that these heuristics are appropriate for our context.

On the other hand, the access-based heuristic does not depend on the amount of contribution. However, the access-based definition is also one of the most basic indicators of core developers. Indeed, the developers who have write access to the main repository have enough knowledge about the product to manage other developers' contributions.

7.2 Internal Validity

Our results for RQ1 are dependent on our heuristics for identifying core developers. In this study, we used 80% of the total contributions as our threshold for identifying core developers, since this threshold was also used by previous studies [5, 6, 8, 17, 18, 24]. While we begin a threshold sensitivity analysis in Section 6, we plan to perform a carefully controlled sensitivity analysis in future work.

Furthermore, our analysis is time-agnostic. Since development teams are changing over time, the number of core developers may vary as well. We plan to conduct a temporal analysis of core teams in future work.

7.3 External Validity

In this study, we filter away projects that have less than 10 developers or less than 1,000 LOC to remove projects that are immature [1, 16]. Therefore, our results may not generalize to legitimate software projects with a small number of contributors.

8. CONCLUSION

Open Source Software (OSS) projects depend heavily on core developers, i.e., team members that produce 80% of the contributions to a project. Prior studies have found that core development teams tend to follow the Pareto principle (a.k.a., the 80-20 rule), i.e., 80% of the contributions are produced by roughly 20% of the contributors. However, these prior studies were performed on small samples of systems. With the recent growth in popularity of the social coding paradigm, a plethora of data is becoming available for researchers to explore core team dynamics within. Therefore, we revisit the analyses of previous work on a large sample of GitHub projects.

To that end, in this paper, we study core development teams on GitHub. Through a case study of 2,496 GitHub projects, we observe that:

- The core teams of many GitHub projects are not compliant with the Pareto principle.
- While some GitHub projects have core teams that are too large to be Pareto compliant, many more have very small core teams, consisting of fewer than 10% of the pool of contributors.
- Core and non-core developers participate in maintenance and future development activities in similar proportions.

9. ACKNOWLEDGEMENTS

This research was partially supported by JSPS KAKENHI Grant Numbers 15H05306, the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Program for Advancing Strategic International Networks to Accelerate the Circulation of Talented Researchers.

10. REFERENCES

[1] T. Bissyande, D. Lo, L. Jiang, L. Reveillere, J. Klein, and Y. le Traon. Got issues? who cares about it? a large scale investigation of issue trackers from github. In *Proc. Int'l Symposium on Software Reliability Engineering (ISSRE)*, pages 188–197, Nov 2013.

[2] V. Cosentino, J. L. C. Izquierdo, and J. Cabot. Assessing the bus factor of git repositories. In *Proc. Int'l Conf. on Software Analysis, Evolution, and Reengineering (SANER)*, pages 499–503, 2015.

[3] K. Crowston, K. Wei, Q. Li, and J. Howison. Core and periphery in free/libre and open source software team communications. In *Proc. Hawai'i Int'l Conf. on System Science (HICSS)*, 2006.

[4] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb. Social coding in github: Transparency and collaboration in an open software repository. In *Proc. Conf. on Computer Supported Cooperative Work (CSCW)*, pages 1277–1286, 2012.

[5] T. Dinh-Trong and J. Bieman. The freebsd project: a replication case study of open source development. *IEEE Trans. on Software Engineering*, 31(6):481–494, June 2005.

[6] J. Geldenhuys. Finding the core developers. In *Proc. of the 36th Euromicro Conference on Software Engineering and Advanced Applications*, pages 447–450. IEEE Computer Society, Sept. 2010.

[7] D. M. German. A study of the contributors of postgresql. In *Proc. Int'l Workshop on Mining Software Repositories (MSR)*, pages 163–164, 2006.

[8] M. Goeminne and T. Mens. Evidence for the pareto principle in open source software activity. In *Joint Porc. the 1st Int'l Workshop on Model Driven Software Maintenance and 5th Int'l Workshop on Software Quality and Maintainability*, pages 74–82, 2011.

[9] G. Gousios. The ghtorrent dataset and tool suite. In *Proc. Int'l Working Conf. on Mining Software Repositories (MSR)*, pages 233–236, 2013.

[10] G. Gousios, M. Pinzger, and A. v. Deursen. An exploratory study of the pull-based software development model. In *Proc. Int'l Conf. on Software Engineering (ICSE)*, pages 345–355, 2014.

[11] G. Gousios, A. Zaidman, M.-A. Storey, and A. v. Deursen. Work practices and challenges in pull-based development: The integrator's perspective. In *Proc. Int'l Conf. on Software Engineering (ICSE)*, 2015. To appear.

[12] L. Hattori and M. Lanza. On the nature of commits. In *Proc. Int'l Conf. on Automated Software Engineering (ASE) - Workshops*, pages 63–71, Sept 2008.

[13] K. Herzig and A. Zeller. The impact of tangled code changes. In *Proc. Int'l Working Conf. on Mining Software Repositories (MSR)*, pages 121–130, 2013.

[14] A. Hindle, D. M. German, and R. Holt. What do large commits tell us?: A taxonomical study of large commits. In *Proc. Int'l Working Conf. on Mining Software Repositories (MSR)*, pages 99–108, 2008.

[15] Y. Jiang, B. Adams, and D. German. Will my patch make it? and how fast? case study on the linux kernel. In *Proc. Int'l Working Conf. on Mining Software Repositories (MSR)*, pages 101–110, May 2013.

[16] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian. The promises and perils of mining github. In *Proc. Int'l Working Conf. on Mining Software Repositories (MSR)*, pages 92–101, 2014.

[17] S. Koch and G. Schneider. Effort, cooperation and coordination in an open source software project: Gnome. *Information Systems Journal*, 12(1):27–42, 2002.

[18] A. Mockus, R. T. Fielding, and J. D. Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Trans. on Software Engineering and Methodology*, 11(3):309–346, 2002.

[19] K. Nakakoji, Y. Yamamoto, Y. Nishinaka, K. Kishida, and Y. Ye. Evolution patterns of open-source software systems and communities. In *Proc. Int'l Workshop on Principles of Software Evolution (IWPSE)*, pages 76–85, 2002.

[20] R. Purushothaman and D. Perry. Toward understanding the rhetoric of small source code changes. *IEEE Trans. on Software Engineering*, 31(6):511–526, 2005.

[21] E. S. Raymond. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1st edition, 1999.

[22] F. Ricca, A. Marchetto, and M. Torchiano. On the difficulty of computing the truck factor. In *Product-Focused Software Process Improvement*, volume 6759 of *Lecture Notes in Computer Science*, pages 337–351. 2011.

[23] G. Robles, J. Gonzalez-Barahona, and I. Herraiz. Evolution of the core team of developers in libre software projects. In *Proc. Int'l Working Conf. on Mining Software Repositories (MSR)*, pages 167–170, May 2009.

[24] G. Robles, S. Koch, J. M. González-Barahona, and J. Carlos. Remote analysis and measurement of libre software systems by means of the cvsanaly tool. In *Proc. the 2nd ICSE Workshop on Remote Analysis and Measurement of Software Systems (RAMSS)*, pages 51–55, 2004.

[25] M. Torchiano, F. Ricca, and A. Marchetto. Is my project's truck factor low?: Theoretical and empirical considerations about the truck factor threshold. In *Proc. Int'l Workshop on Emerging Trends in Software Metrics (WETSoM)*, pages 12–18, 2011.

[26] Y. Ye and K. Kishida. Toward an understanding of the motivation open source software developers. In *Proc. Int'l Conf. on Software Engineering (ICSE)*, pages 419–429, 2003.

Software Evolution and Time Series Volatility: An Empirical Exploration

Jukka Ruohonen
Department of Information Technology, University of Turku, FI-20014 Turun yliopisto, Finland
juanruo@utu.fi

Sami Hyrynsalmi
Department of Information Technology, University of Turku, FI-20014 Turun yliopisto, Finland
sthyry@utu.fi

Ville Leppänen
Department of Information Technology, University of Turku, FI-20014 Turun yliopisto, Finland
ville.leppanen@utu.fi

ABSTRACT

The paper presents the first empirical study to examine econometric time series volatility modeling in the software evolution context. The econometric volatility concept is related to the conditional variance of a time series rather than the conditional mean targeted in conventional regression analysis. The software evolution context is motivated by relating these variance characteristics to the proximity of operating system releases, the theoretical hypothesis being that volatile characteristics increase nearby new milestone releases. The empirical experiment is done with a case study of FreeBSD. The analysis is carried out with 12 time series related to bug tracking, development activity, and communication. A historical period from 1995 to 2011 is covered under a daily sampling frequency. According to the results the time series dataset contains visible volatility characteristics, but these cannot be explained by the time windows around the six observed major FreeBSD releases. The paper consequently contributes to the software evolution research field with new methodological ideas, as well as with both positive and negative empirical results.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*Process metrics*

General Terms

Economics, Experimentation, Measurement

Keywords

Software evolution, code churn, time series analysis, volatility, conditional variance, ARIMA, GARCH, FreeBSD

1. INTRODUCTION

This exploratory empirical paper has a dual purpose (P): to introduce and evaluate time series volatility modeling for software evolution research (P_1), and to investigate whether volatility increases near new operating system releases (P_2).

Volatility is part of the econometric time series nomenclature. While no unequivocal definitions can be given for the phenomenon, some aspects of volatility are generally agreed upon particularly in financial econometrics; volatility is related to risks, uncertainty, and variance. Accordingly, variability depends on past variability, which causes uncertainty and risks in planning and estimation. In stock markets a conventional reasoning starts from the serially correlated arrival of new and unanticipated news available to the traders [1, 2]. The actors and their algorithms will have different reactions to these news; someone or something may sell, someone may buy. This causes rapid short-run fluctuations. As someone or something may subsequently buy or sell as a response to the increased activity, volatility tends to cluster in time. Once the buyers and sellers have amalgamated the new news and each other's reactions, trading returns to its normal trajectory. The message from this naïve sketch for an economic interpretation is that the interest in volatility modeling is to observe activity, volatility, rather than the time series averages, or the direction of time series.

Volatility is likely to be observable in numerous different software evolution applications [3], although the generative theoretical mechanisms differ. It can be related to the modeling of email flows, for instance. If *volume* measures the number of emails, *velocity* would measure how rapidly the volume of emails changes [4]. This is also one theoretical rationale for volatility modeling: early resolution of uncertainty helps to plan the future, but the point in volatility modeling is not necessarily whether the release of information will result changes, but rather whether the acceleration in the flow of information will result changes [5]. If change rates, or first-differences, $\Delta y_t = y_t - y_{t-1}$, are observed, increasing volatility would then indicate increasing velocity through increasing variance in the change rates. The econometric volatility effect is also stochastic; a large change in volume is likely to be followed by another large change.

Perhaps the most apparent long-run software evolution explanation relates to new milestone releases around which software engineering and related activities tend to increase or intensify [6, 7]. Schedules are set, final development iterations are completed, increasing variability is introduced by branching [8], reliability assessments are made, marketing scenarios are planned or fine-tuned, and so forth. Since code *churn* presumably increases temporally as a result, the probability of making mistakes likely increases, including the like-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

IWPSE'15, August 30, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3816-5/15/08...$15.00
http://dx.doi.org/10.1145/2804360.2804367

lihood to introduce security vulnerabilities [9, 10, 11]. Besides these software engineering aspects, volatility may be important for marketing purposes. In the age of crowdsourcing and fundraising, knowing when a crowd reacts may be important in targeting the timing of marketing and other visibility campaigns, for instance.

The traditional software evolution background would also relate to the continuous changes, which should not be too rapid in order to maintain developers' ability to absorb the new information related to a code base [12]. As will be later elaborated in detail, volatility tends to cluster in time. In a sense, therefore, volatility works against the principle of constant and stable change rates. The effect may be seen in different short-run sliding windows that measure larger commit transactions in version control systems [13]. Software engineering processes offer another example. As volatility implies time-dependencies in the variance of a time series, common practices such as sprints in agile development are likely to cause different volatility characteristics; variance may start to cluster towards the end of a sprint, for instance.

In a summary, there are numerous different software engineering aspects that may lead to time series volatility. While the paper leaves the theoretical interpretation open, the empirical rationale is clear: an empirical baseline is required for volatility modeling before explicit, theoretically meaningful software engineering questions can be investigated in more detail (P_1). Nevertheless, the timing of major operating system releases is used to theoretically motivate the empirical experiment that models volatility in twelve high-frequency time series extracted from the FreeBSD development infrastructure (P_2). The expectation is that volatility increases in the proximity of the six major FreeBSD releases made between 1995 and 2011. Before proceeding to the actual experiment, the empirical approach is first elaborated.

2. EMPIRICAL APPROACH

The empirical approach can be elaborated by first considering the basic time series rationale behind econometric volatility modeling. The proximity of releases is subsequently discussed in relation to the models. The fitting strategy follows.

2.1 Volatility

The time series volatility phenomenon is best illustrated by considering simple algebraic assumptions in time series regression models. Therefore, consider a model:

$$y_t = f(y_t \mid \mathcal{I}_{t-}) + \epsilon_t = \mathbf{x}_t'\boldsymbol{\beta} + \epsilon_t, \quad t = 1, \ldots, T, \tag{1}$$

where y_t is the dependent (modeled) time series, $\mathbf{x}_t'$ is a vector of independent variables, $\boldsymbol{\beta}$ is a vector of regression coefficients, and ϵ_t is the residual process. The function $f(\cdot)$ is used to denote an arbitrary time series model that is based on the information set, $\mathcal{I}_{t-}$, available to model estimators before the current time index t. The vector $\mathbf{x}_t'$ may, therefore, contain also lagged values of y_t, among other explanatory variables.

For instance, and for the purposes of this paper, the function $f(\cdot)$ can be assumed to denote the classical autoregressive (AR) integrated (I) moving-average (MA) model (ARIMA). Given non-negative integers p, d, and q for the order of the abbreviations AR, I, and MA, respectively, the general goal is to have a small parsimonious ARIMA(p, d, q) model that renders ϵ_t as white noise while providing good predictive power. While further acronyms and different alterations are often required, this classical model generally performs even amazingly well in forecasting [6, 14]. The empirical rather than theoretical motivation behind the ARIMA model is illustrative also from a different viewpoint.

The goal in ARIMA-like modeling is to forecast the future. In the context of the regression equation (1) this implies that the model predicts the *conditional mean*,

$$\mathrm{E}(y_t \mid \mathbf{x}_t') = \mathbf{x}_t'\boldsymbol{\beta}, \quad \mathbf{x}_t' \in \mathcal{I}_{t-}. \tag{2}$$

If the model fits well for a given dataset, assumptions

$$\mathrm{E}(\epsilon_t) = 0, \quad \mathrm{Var}(\epsilon_t) = 1, \quad \text{and} \quad \epsilon_t \sim \mathrm{N}(0, 1) \tag{3}$$

are often made. In other words, ϵ_t is independent and identically distributed from the normal distribution with a mean of zero and variance equal to unity; the residual process is Gaussian white noise. Time series volatility models are interested in the cases for which these assumptions do not apply. More specifically, the interest is to model the conditional variance rather than the conditional mean.[1]

Consider now that the model in (1) instead yields a residual process υ_t that contains the white noise ϵ_t from (3), and another component that is dependent on the past:

$$\upsilon_t = \epsilon_t \sigma_t, \tag{4}$$

where σ_t denotes the positive time-varying component of interest [15]. This provides the underlying idea in volatility modeling: the goal is to condition this component based on the information available in $\mathcal{I}_{t-}$. In terms of (4) this means that the white noise process is multiplied by the conditional standard deviation, σ_t, which is always non-negative.[2]

Thus, from a Bayesian viewpoint, the idea is to make dependent draws from the normal distribution rather than independent draws from $\sim \mathrm{N}(0, 1)$, that is, from the standard normal distribution. Although normality as such is not important, this amounts to saying that

$$\upsilon_t \mid \mathcal{I}_{t-} \sim \mathrm{N}(0, \sigma_t^2) \tag{5}$$

or that

$$y_t \mid \mathcal{I}_{t-} \sim \mathrm{N}[\mathbf{x}_t'\boldsymbol{\beta}, \mathrm{Var}(y_t \mid \mathcal{I}_{t-})], \tag{6}$$

where $\mathbf{x}_t'\boldsymbol{\beta}$ is the conditional mean from (2). In other words, both the conditional mean and the conditional variance are parameterizable functions of the information in $\mathcal{I}_{t-}$.

The statistical meaning of volatility can be now contemplated by comparing the assumption $\sim \mathrm{N}(0, 1)$ in (3) to the volatility assumption in (5). Since the residual process from (1) now has a variance σ_t^2 that can be predicted by the available information before t, the conditional variance evolves in time. For instance, the variance might be dependent on the past variances at $t - 1$ and $t - 2$,

$$\mathrm{Var}(\upsilon_t \mid \upsilon_{t-1}, \upsilon_{t-2}) = \sigma_t^2, \tag{7}$$

or more generally

$$\sigma_t^2 = \mathrm{Var}(\upsilon_t \mid \mathcal{I}_{t-}) = \mathrm{Var}(y_t \mid \mathcal{I}_{t-}). \tag{8}$$

[1] In terms of the existing classification of software evolution volatility into amplitude, periodicity, and dispersion [3], econometric volatility modeling would concentrate to the last type. Note, though, that for instance periodicity can be incorporated to the econometric volatility models.

[2] Notice that the unconditional mean of υ_t still remains constant; taking expectations from (4) yields zero due to the white noise assumptions in (3).

An intuitive interpretation would be that in case the past drews, $\sim \mathrm{N}(0, \sigma^2_{t-1})$ and $\sim \mathrm{N}(0, \sigma^2_{t-2})$, were done by specifying a large dispersion for the normal probability distribution, then also σ^2_t is likely to be large. This explains why volatility tends to cluster in time. This tendency is neither deterministic nor infinite, however. Although the dependence on time typically fades away quickly, a large variance does not translate always to a large variate [16]. In other words, even if σ^2_{t-2} and σ^2_{t-1} were large, the variance at t may sometimes be moderate. Nevertheless, a time series *shock* – such as a new release – will show as a large deviation of the dependent process from the specified conditional mean (i.e., as a deviation of y_t from the fitted values), but the same shock can be seen also as a large positive or negative value in the residual process [17]. Volatility models do not make assumptions about the signs of the residuals, however. In other words, large values in the residual process tend to be followed with large values of either sign.

This statistical meaning also underlines the theoretical weaknesses in volatility modeling; what does it mean to model the conditional variance? In financial econometrics volatility modeling became extremely popular supposedly largely because of direct theoretical applicability and value in practical policy decisions. In other fields the formulation of theory around the processes has often been more difficult. One way to approach the question is technical. It has been suspected, among other things, that the processes such as (7) might be simply due to the effect of variables omitted from estimated models [15]. This portrays the processes in usual statistical terms; as evidence of misspecification and nuisance to get rid of. In software evolution this rationale would perhaps lead to smoothing with different time series filters [6, 18], given the general empirical software engineering rationale to clean datasets from undesirable noise [19], and to avoid the omitted variable bias [20]. If the volatility noise is retained, however, it is often unclear in applied work whether the theoretical shocks should be modeled in terms of the conditional mean or the conditional variance. The empirical experiment follows the latter path; the new major releases are assumed to shock the second conditional term in (6). If this theoretical hypothesis is rejected, the potential volatility characteristics (P_1) are likely generated by other mechanisms than release engineering and associated activities (P_2). If the hypothesis holds, the concept of *time deformation* [1, 21] could be adopted as a high-level theoretical explanation; the real calendar time and the release-cycle pseudo-time may proceed at different speeds, generating different volatility characteristics.

2.2 The Proximity of Releases

Consider the following simple definition for a deterministic control variable PRX^r_t, the proximity of the r:th release:

$$\mathrm{PRX}^r_t(t_a, t_r, t_b) = \begin{cases} 1 & \text{if} \quad t_a \leq t_r \leq t_b \\ 0 & \text{if} \quad t_r \notin [t_a, t_b] \end{cases}, \tag{9}$$

where all three date indices are inside a time series sample window, and t_r denotes an index of a given release day. A case $t_a = t_r = t_b$ defines a standard dummy variable, omitting the word proximity present in the abbreviation. That is, only the actual release day would be observed.

The software engineering motivation for (9) comes from the sliding window model that was initially introduced for the detection of commit transaction sequences and associated bursts [13], and then extended to other purposes with good results [22, 23]. The general idea is also rather similar to the windows used in the so-called event studies that examine stock market reactions to different unanticipated events, including, for instance, mergers and acquisitions, computer security breaches, and outsourcing events and announcements [2, 24]. These have been incorporated also to volatility modeling [1]. Regardless of the domain of application, there is a generic problem associated with all simple, deterministic windows: the length cannot be easily defined. While different transition state models [25] could be considered, the problem can be approached also theoretically.

Although large variations are present between different open source projects, existing empirical research indicates that traffic increases near adoption peaks (releases), while traffic in development-oriented systems increases before releases or large refactoring periods [26]. The latter effect is typical particularly in the BSD operating system context within which releases are polished for a relatively long period by the developer community, although a release engineering team typically handles the actual scheduling and branching. The preparation periods include, for instance, different "hackathons", which hint that commits and general programming activity increase before releases. However, provided that adoption peaks nearby major releases, also systems such as bug tracking should show increasing volatility as bugs are discovered by the newly arrived users. Further examples are easy to pinpoint to signify this heterogeneity assumption. For instance, the arrival of social media has partially challenged the existing mailing list communication media; open source blogging has been observed to increase after releases, partially again because of the non-developer communities [7]. All in all, it seems that volatility is likely to vary from a software development work system to another, but it is more difficult to make clear-cut theoretical corollaries regarding the timings.

To make the theoretical assumptions explicit: the window in (9) is defined with a positive integer that is uniformly shared between the observed $r = 1, \ldots, R$ releases. That is, a *modified proximity metric* $\tilde{\mathrm{PRX}}^r_t$ takes a value one in case $t_r \in [t_r - k, t_r + k]$ for some shared positive integer $k > 0$, $t_r + k < T, t_r - k > 1$. For instance, with $k = 1$ a metric $\tilde{\mathrm{PRX}}^r_t(k, t_r, k)$ yields a vector $(0, \ldots, 0, 1, 1, 1, 0, \ldots, 0)$, which would be presumed to capture the shock introduced by the r:th release in the conditional variance, given a model for the conditional mean.

2.3 Estimation

The seminal work in [15] introduced an autoregressive conditional heteroskedasticity model, denoted by ARCH(q). The key idea behind the ARCH(q) model is that υ_t in (4) is uncorrelated across time with $\mathrm{E}(\upsilon_t) = 0$, but the conditional variance of it, σ^2_t, may change in time. The original model required weights to ensure that volatility will eventually decay. This motivated the generalized ARCH representation known as GARCH. This GARCH(p, q) model, which was introduced in [27], is based on a seemingly ARMA-like idea:

$$\sigma^2_t = \alpha_0 + \sum_{i=1}^{q} \alpha_i \upsilon^2_{t-i} + \sum_{j=1}^{p} \beta_j \sigma^2_{t-j}, \tag{10}$$

where $q > 0$, $p \geq 0$, $\alpha_0 > 0$, and $\alpha_i \geq 0$ for all $i = 1, \ldots, q$, and $\beta_j \geq 0$ for all $j = 1, \ldots, p$. These restrictions en-

sure that the conditional variance is always strictly positive. An important mathematical result relates to the unconditional variance that should be asymptotically constant in order for the process to be stationary [15, 27]. For instance, given the non-negativity restriction required for variance, a GARCH(1, 1) process is (weakly) stationary if $\alpha_1 + \beta_1 < 1$.

The model building process is similar to ARIMA modeling. In fact, the volatility models are typically fitted by using an ARMA structure for the mean and a GARCH structure for the variance, resulting GARCH(p_1, q_1)-ARIMA(p_2, d, q_2) models. The identification is often easier for the variance parameters p_1 and q_1 than for the conventional AR(p_2) and MA(q_2) terms. For instance, it has been observed that a short GARCH(1, 1) structure is adequate to represent many financial time series [17]. This was used as the reference point in the empirical experiment. The ARMA structures were constructed by visually investigating correlograms, and by checking that no remaining autocorrelation was present according to the Ljung-Box test. The (G)ARCH effects can be evaluated with the Engle-test named after the inventor. In essence: given (10) and residuals from (1), a model

$$\widehat{\epsilon}_t^2 = \alpha_0 + \alpha_1 \widehat{\epsilon}_{t-1}^2 + \alpha_2 \widehat{\epsilon}_{t-2}^2 + \cdots + \alpha_q \widehat{\epsilon}_{t-q}^2 + u_t, \qquad (11)$$

should fit well in case there are volatility characteristics in y_t. Alternatively, the Ljung-Box test can be used to investigate the effect of the past squared residuals.

There are at least two approaches to model the release windows. The first is based on the conventional machine learning principle: given a predefined period before the t_r:th release, train the data in a window before t_r, and then carry out the evaluation in the window starting from $t_r + k$.[3] This has been also a common principle in the noted event studies [28], although in time series analysis numerically equivalent results can be (often) obtained by using deterministic dummy variables [29, 30]. The latter path is suitable for the purposes of this paper. Consequently, the modified $\tilde{\mathrm{PRX}}_t^r$ metric was first used to investigate the effect of each of the six major releases separately. The results were similar to using the following additional simplification:

$$\widetilde{\widetilde{\mathrm{PRX}}}(k) = \sum_{j=1}^{6} \tilde{\mathrm{PRX}}^{r_j}(k, t_{r_j}, k), \qquad (12)$$

which aggregates the individual effects into a single vector. This is used in the results reported. The following windows were tested: $k = 1$ (one day before and after), $k = 3$ (a week), and $k = 15$ (approximately a month). The results were similar with all integers. Finally, the *rugarch* package [31] is used for computation.

3. EMPIRICAL EXPERIMENT

The empirical experiment proceeds by first introducing the FreeBSD dataset. A few statistical observations are subsequently made. The volatility estimates follow.

3.1 Data

The twelve time series metrics are summarized in Table 1. All series are observed daily during a period from the first day of January 1995 to the last day of December 2011. Following the existing practices [32], all metrics were collected on *a priori* grounds, meaning that the selected features were chosen before analysis. This choice also limited the end of the sampling period to 2011. The time series metrics are grouped into four categories: bug tracking, mailing lists, commit activity, and commit magnitudes.

Two major changes have been made in the FreeBSD project regarding the essential software development tools. First, the Concurrent Versions System (CVS) was replaced with Subversion in 2008. All data was extracted from the transformed Subversion histories. Second, the project decided to abandon the old email-based GNATS bug tracking system in 2014. The bug tracking data refers only to the historical tracking with the old system. The two metrics BRA and BRC, the number of arrived and closed bug reports, respectively, are based directly on data extracted from GNATS. No classifications are mare according to the status, type, or category of bugs. The third bug tracking metric, BRR, records the number of replies made to the reports. Given the way GNATS works [33], BRR was matched from replies to the `freebsd-bugs` mailing list by using a regular expression.

Mailing lists provide the primary medium for epistemic interactions [34] particularly in the open source operating system context. Four mailing list archives were consequently included in the sample: besides the noted `freebsd-bugs`, `freebsd-current` (discussions concerning the running development branch), `freebsd-hackers` (general technical discussions and debates), and `freebsd-questions` (user questions and support) are observed. These refer to BRR, MLC, MLH, and MLQ in Table 1, respectively.

Development activity is observed with six metrics grouped into two categories, both of which relate to code churn [11]. First, the three commit activity metrics simply count the number of commits. The three activity metrics (CTB, CTP, and CTD) differ in their domain, however. FreeBSD contains three separate development domains with three separate version control trees: `base` refers to the actual BSD operating system; `ports` is used for packing external third-party open source software; and `doc` is used for documentation. Second, the analogously grouped metrics LDB, LDP, LDD were obtained commit-by-commit with `svnlook diff`, which was passed to `wc -l`. Consequently, the metrics measure *roughly* the total size or magnitude of commits in terms of the changed files and lines in a day. These are, of course, only approximations; there is a large and systematic divergence from the magnitude of changed source code already due to the additional information printed by Subversion. Since the empirical interest (P_2) is to model volatility – for which churn seems like the ideal case – the interpretations should not be significantly threatened by these inaccuracies.

All collected metrics were aggregated to cumulative daily sums. In the time series literature this aggregation solution is sometimes referred to as a *flow* scheme, as opposed to a *stock* or aggregation by using different averages [35]. More importantly, the time series are not irregular but instead follow equally spaced, daily intervals in calendar time. Given four leap years, each time series $i = 1, \ldots, 12$ has a length of $T = (4 \times 366) + (13 \times 365) = 6209$. No data transformations were applied, although the three commit magnitude metrics

[3] Although the estimation can be done for instance by considering the estimation window, the event window, and the post-event window [28], it should be emphasized that the time series context forbids randomization of observations in the windows. That is to say, the approach should not be confused with the k-fold cross-validation techniques.

Table 1: Metrics

1. Bug Reports		3. Commit Activity	
BRA	Number of arrived bug reports	CTB	Number of commits to `base`
BRC	Number of closed bug reports	CTP	Number of commits to `ports`
BRR	Number of replies to bug reports	CTD	Number of commits to `doc`
2. Mailing Lists		**4. Commit Magnitudes**	
MLC	Number of messages on `current`	LDB	Lines of commit differences in `base`
MLH	Number of messages on `hackers`	LDP	Lines of commit differences in `ports`
MLQ	Number of messages on `questions`	LDD	Lines of commit differences in `doc`

The bug tracking and mailing list data is based on archives that were provided by the FreeBSD project in July 2014 from the server at `ftp://ftp.freebsd.org`. The commit data is based on full Subversion trees provided from the same server for the purposes of mirroring.

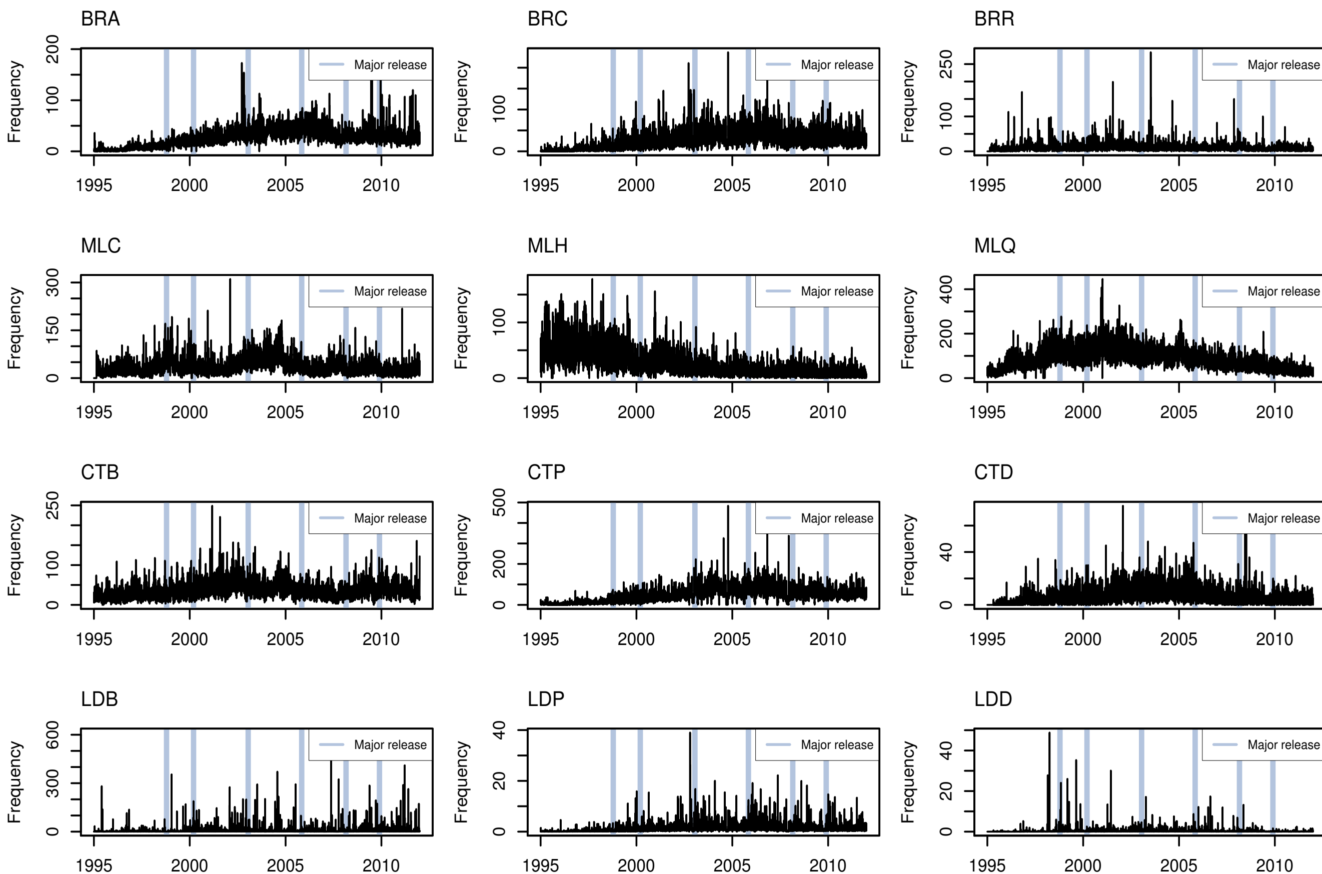


Figure 1: Data

were multiplied by 10000^{-1} for easier interpretation. This has no effect on the results reported.

3.2 Preliminaries

The metrics are visualized in Fig. 1. In general, the laws of software evolution [12] have been observed to hold also in FreeBSD. The continuous growth of the code base has been observed to follow a linear trend, for instance [36]. The six commit activity and magnitude metrics generally support the observation; the development activity has not halted. Most of the time series show significant volatility noise, however, which prevents a human eye from clearly seeing the time series trends. If the kernel smoothing techniques [6] are used to aid the visual inspection, several different trends become more visible. For instance, the trend in MLQ shows an interesting boomerang-shape. Another point worth remarking relates to the trends in the bug tracking system (see Fig. 2). The arrival of new bug reports seems to have closely followed the closure of old reports in the long-run.

On the other hand, even after filtering a human eye has difficulties in seeing a clear clustering pattern around the six major releases. There are some peaks, however. For instance, some series peak near the fifth major release. This historical release was developed in the first part of the 2000s

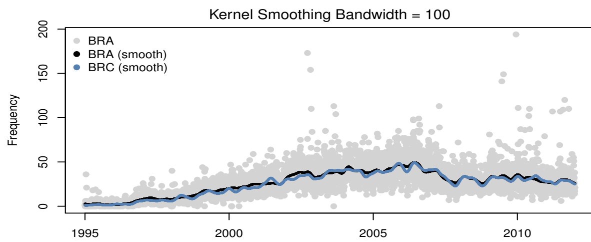


Figure 2: Trends in BRA **and** BRC

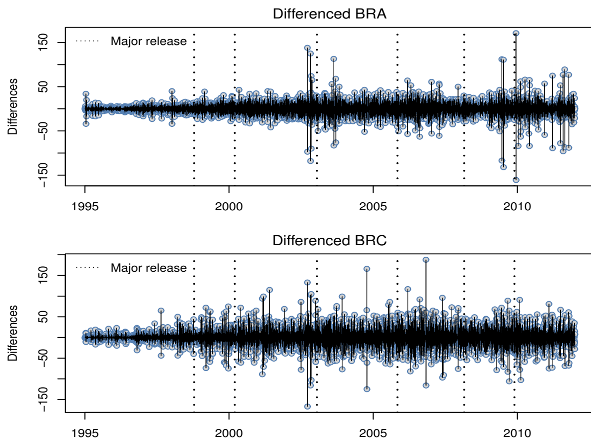


Figure 3: Differenced BRA **and** BRC

during which symmetric multiprocessing was introduced to FreeBSD, among other large architectural changes that also caused some community problems [37]. In general, however, the visible turbulence periods do not seem to systematically cluster around the proximity of all observed releases.

Other basic time series characteristics can be summarized with three general observations. First, volatility models assume *stationarity*. Formal test results are consequently reported in Table 2. The null hypothesis of stationarity is clearly rejected for all series, while all differenced series appear to be stationary. A further concern relates to the visible level shifts in the variance trajectories (see Fig. 1), which are typical factors that may undermine the stationary assumption [16], leading to the so-called integrated I-GARCH model [38]. Nevertheless, the analysis is carried out under the assumption that the first-differenced series are adequate for the fitted GARCH-AR(1)MA models. To illustrate the graphical shape of the first-differenced series, the differenced BRA and BRC are shown in Fig. 3.

Second, *normality* is a potential problem. As might be typical for software engineering time series, all differenced series are distributed according to a bell-shaped curve that somewhat resembles the normal distribution, although the mass is sharply located around zero. Formal tests, consequently, reject normality for all univariate series, whether in levels or in differences (see Table 3). This suggests that some alternative distribution could be preferable. In volatility modeling the Student's t-distribution has been a popular alternative for the normal distribution [16, 31]. It also seems like a reasonable choice to describe the density shapes of the differenced series. To investigate this possibility further, Fig. 4 shows quantile-quantile plots for the differenced BRA and BRC using the (central) Student's t-distribution. A visual conclusion confirms that the normality assumption may not be realistic. Consequently, the t-distribution is specified for the reported models, although the same conclusions can be reached under the more familiar normal distribution.

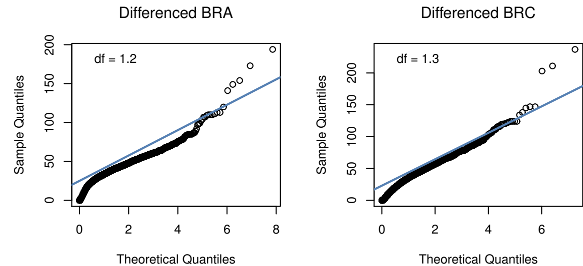


Figure 4: Two Student's t Q-Q Plots

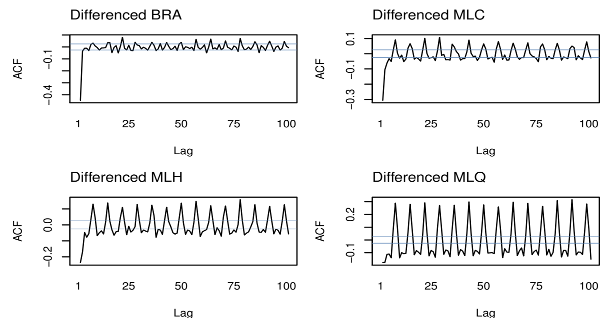


Figure 5: Four Correlograms

Third, the three mailing list time series exhibit *seasonality*. This can be illustrated with the four correlograms in Fig. 5. (All other series show similar patterns to the shown differenced BRA). The cycle seems to follow relatively clearly the weekly variation. While seasonal ARIMA models [14] could be considered for the conditional mean models, no special seasonal effects are included for the three mailing list metrics.[4] The autocorrelation functions (ACFs) also indicate that the memory of the differenced series seem to be relatively short in general. Moreover, differencing seems to cause a negative autocorrelation at the first lag, suggesting that simple first-differences do not fully account for the trending characteristics in the level of the series.

3.3 Results

The formal modeling can be started by investigating the time-dependence of the residual variance processes. This can be done in two simple steps. First, plain AR(p) models were fitted for the first-differenced series by letting the Akaike's information criterion (AIC) to pick the suitable lag length automatically. Second, the squared residuals from the autoregressive results were used to estimate (11) by again

[4] The reason is partially practical; the used implementation [31] does not allow to easily define seasonal components.

Table 2: Unit Root Tests (KPSS)

	Levels		Differences			Levels		Differences	
	Value	Result	Value	Result		Value	Result	Value	Result
BRA_t	11.42	$\sim I(1)$	0.01	$\sim I(0)$	CTB_t	2.25	$\sim I(1)$	0.02	$\sim I(0)$
BRC_t	11.18	$\sim I(1)$	0.01	$\sim I(0)$	CTP_t	11.76	$\sim I(1)$	< 0.01	$\sim I(0)$
BRR_t	1.20	$\sim I(1)$	0.01	$\sim I(0)$	CTD_t	3.84	$\sim I(1)$	0.01	$\sim I(0)$
MLC_t	1.52	$\sim I(1)$	0.01	$\sim I(0)$	LDB_t	3.33	$\sim I(1)$	< 0.01	$\sim I(0)$
MLH_t	14.46	$\sim I(1)$	0.01	$\sim I(0)$	LDP_t	7.72	$\sim I(1)$	< 0.01	$\sim I(0)$
MLQ_t	5.59	$\sim I(1)$	0.02	$\sim I(0)$	LDD_t	1.18	$\sim I(1)$	< 0.01	$\sim I(0)$

The table reports results from the so-called KPSS stationarity tests [39]. The null hypothesis is a stationary process, denoted by $\sim I(0)$. The alternative of non-stationarity is marked with $\sim I(1)$. More specifically: a tested series x_t is assumed to follow $x_t = z_t + u_t$, where $z_t = z_{t-1} + \epsilon_t$ and $\epsilon_t \sim \text{N}(0, \sigma_\epsilon^2)$. Given this representation, H_0 is that $\hat{\sigma}_\epsilon^2 = 0$, implying that the additional pure random walk term, z_t, is constant around zero mean. The ancillary lag length was truncated from $4 \times \sqrt[4]{(T/100)}$ to 33. Critical values at ten, five, and one percent levels are 0.35, 0.46, and 0.74, respectively, as given by the `ur.kpss` function used for estimation [40].

Table 3: Normality Tests (Shapiro-Wilk)

	Levels		Differences			Levels		Differences		
	Value	p	Value	p		Value	p	Value	p	Density of ΔCTB_t
BRA_t	0.95	✓	0.87	✓	CTB_t	0.91	✓	0.96	✓	
BRC_t	0.90	✓	0.92	✓	CTP_t	0.92	✓	0.86	✓	
BRR_t	0.61	✓	0.75	✓	CTD_t	0.83	✓	0.90	✓	
MLC_t	0.88	✓	0.93	✓	LDB_t	0.24	✓	0.38	✓	
MLH_t	0.84	✓	0.95	✓	LDP_t	0.53	✓	0.60	✓	
MLQ_t	0.95	✓	0.98	✓	LDD_t	0.14	✓	0.24	✓	

The table shows results from the Shapiro-Wilk test; the symbol ✓ marks cases in which H_0 of normality is rejected at 0.001 level. Due to limitations in R's `shapiro.test`, each result is an average from a hundred random samples of size 5000 (from $T = 6209$). The densities of all differenced metrics are roughly similar to the shown density of the differenced amount of commits made to `base`.

letting AIC to choose the lag length. The results are summarized in Fig. 6. While it is clear that a large p is required to describe the differenced series (the left-hand side plot), many metrics indicate also rather long AR(p) processes for the squared residual terms. Evaluating the estimated AR coefficients reinforces the visual conclusion that there are (G)ARCH effects present in most of the time series.

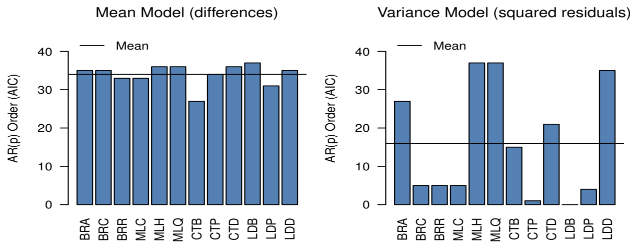


Figure 6: Coarse Volatility Effects

The fitted models are summarized in Table 4. Relatively short GARHC(p_1, q_1) orders are required to control the process in (10) such that no remaining effects are present. Only a few series required higher orders than $p_1 = q_1 = 1$ according to the Engle-test. This is a familiar finding from financial econometrics [1, 17]. Also decent ARMA(p_2, q_2) structures can be located relatively effortlessly by trial-and-error. Most of the fitted models are free of remaining autocorrelation.

However, the seasonality patterns affect the mailing list estimates for which decent ARMA structures are difficult to formulate to capture the time series characteristics of MLC, MLH, and MLQ. If information criteria measures and parsimonious models are used to rank the estimates, the winner is a model for LDD.

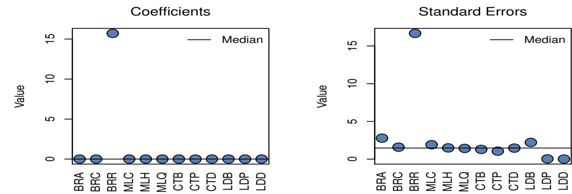


Figure 7: Effect of Release Windows ($k = 3$)

Alas, the proximity of releases has no explanatory power. The coefficients are negligible and the standard errors are large for all but one series (see Fig. 7). The only exception is BRR for which both a very high coefficient and a very high standard error are present. The aggregated window is not statistically significant ($p \simeq 0.35$), however. Widening or shortening the one week windows do not change the results. The same applies to the disaggregated $\tilde{\text{PRX}}_t^r$ windows. It is highly unlikely that the volatility effects can be explained by the proximity of the six major FreeBSD releases (P_2).

Table 4: Volatility Model Diagnostics (GARCH-ARIMA)

	BRA	BRC	BRR	MLC	MLH	MLQ	CTB	CTP	CTD	LDB	LDP	LDD
GARCH(p_1, q_1)	(2, 1)	(1, 1)	(1, 1)	(1, 1)	(2, 1)	(1, 1)	(1, 1)	(1, 1)	(1, 1)	(1, 1)	(1, 1)	(0, 1)
ARIMA(p_2, 1, q_2)	(9, 9)	(10, 10)	(2, 2)	(2, 5)	(6, 6)	(10, 10)	(8, 8)	(8, 2)	(8, 2)	(3, 4)	(6, 4)	(1, 1)
Ljung-Box(1)	–	–	–	–	–	–	–	–	–	–	–	–
Ljung-Box(a)	–	–	✓	✓	✓	✓	–	–	–	–	–	–
Ljung-Box(b)	–	–	–	✓	✓	✓	–	–	–	–	–	–
Engle($p_1 + q_1 + 1$)	–	–	–	–	–	–	–	–	–	–	–	–
Engle($p_1 + q_1 + 3$)	–	–	–	–	–	–	–	–	–	–	–	–
Engle($p_1 + q_1 + 5$)	–	–	–	–	–	–	–	–	–	–	–	–
BIC	6.96	7.75	6.69	8.42	7.68	9.09	8.24	8.54	5.38	5.09	2.04	-0.85
No. insignificant	2	2	1	1	2	1	1	2	3	1	2	1
Subjective evaluation	A	A	B	C	C	C	A	A	A	A	A	A

A symbol ✓ denotes $p < 0.05$ in the Ljung-Box test for no autocorrelation (H_0) and in the Engle-test for no remaining ARCH effects (H_0). In both tests the tested lag is shown in parenthesis, given the orders shown in the first two rows. In the former test the lags a and b are defined in [31] as $a = 2 \times (p_2 + q_2) + (p_2 + q_2) - 1$ and $b = 4 \times (p_2 + q_2) + (p_2 + q_2) - 1$. The subsequent row shows the Bayesian information criterion (BIC), followed by the total number of statistically insignificant parameters at 5 % level (using normal, non-robust standard errors). The final row denotes a subjective evaluation according to a threefold criteria: decent (A), moderate (B), and bad (C). All models include the aggregated deterministic variable from (12) with $k = 3$ in the variance (GARCH) model. The models are specified with the Student's t-distribution. The constant term is included in both the mean models and the variance models, although it is insignificant in some models.

The empirical experiment can be completed by demonstrating the predictive potential. The following seems like a theoretically decent example model that might be of interest also to the FreeBSD developers and their bug tracking efforts:

$$\Delta\mathrm{BRC}_t = \mathbf{x}_t'\boldsymbol{\beta} + \epsilon_t, \quad \text{where} \tag{13}$$

$$\Big[\beta_1\Delta\mathrm{BRA}_{t-1}, \beta_2\Delta\mathrm{BRR}_{t-1}, \beta_3\Delta\mathrm{CTB}_{t-1}, \beta_4\Delta\mathrm{LDB}_{t-1}, \beta_5\widetilde{\mathrm{PRX}}_t(3)\Big] \in \mathbf{x}_t'\boldsymbol{\beta}$$

among past information about $\Delta\mathrm{BRC}_t$. Since no additional information from $\mathcal{I}_{t-}$ is used for the (exogenous) independent variables, the interpretation is relatively clear: the model predicts the current daily changes in closed bug reports by the past changes in arrived bug reports, the past churn in the amount of bug report replies, and the past changes in the development activity within the `base` operating system domain, controlling for the proximity of releases with a weekly window for each release. The direction of causality is not clear, however. For instance, also $\Delta\mathrm{BRA}_t$ could appear on the left-hand side since general development churn is likely to increase the changes in the amount of opened bug reports. Nonetheless, in addition to forecasting, a time series model such as (13) can be used to investigate the factors that affect the closure (or arrival) of bug reports, and to evaluate whether these effects are stable and systematic across time. Since the two metrics BRA_t and BRC_t follow similar long-run trends (see Fig. 2), the general idea for (13) could also be that the two basic bug tracking metrics are intentionally kept synchronized.

The relatively high-order specification for $\Delta\mathrm{BRC}_t$ in Table 4, $p_1 = q_1 = 1$ and $p_2 = q_2 = 10$, can be used to estimate the model. When the additional series are included in the conditional variance equation in (10), the effects are negligible and statistically insignificant. The BIC is 7.72. When the four metrics are instead used in the more conventional conditional mean model, $\widehat{\beta}_1$ and $\widehat{\beta}_3$ are significant at 5 % level with positive signs. Also BIC reduces slightly to 7.69. In other words, some dynamic relationships are present particularly with respect to BRC and BRA, although the effects do not show in the variance trajectories. Regardless whether the additional five metrics are included in the mean model (M_1) or in the variance model (M_2), the sum of the two estimated GARCH coefficients is rather close to unity, suggesting further model calibration to guard for potential violations regarding the stationarity assumption.

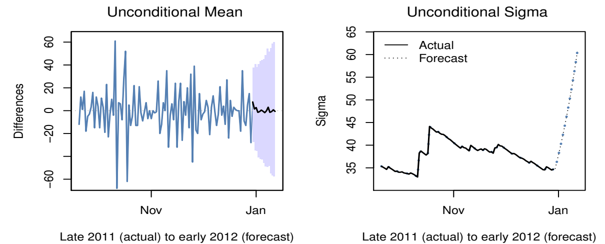


Figure 8: Forecasts (M_1) with (13)

Nevertheless, the underlying forecast setup can be illustrated in the form of Fig. 8, which shows an example of a two week out-of-sample (2012) forecast based on unconditional expectations. As can be seen, the unconditional mean is predicted to remain relatively constant. As the forecast window widens, however, the unconditional expectation for the sigma in (4) starts to increase, indicating increasing uncertainty. This illustrates the practical relevance of volatility modeling: both empirical accuracy and theoretical questions can be approached also by examining the variables and patterns that influence σ_t, the conditional standard deviation.

4. CONCLUSION

It has been widely acknowledged that econometric volatility modeling is only a mechanical way to describe and predict the variance behavior of a time series [15, 16]. It does not provide any explanation on why the variance characteristics tend to rather neatly follow the relatively simple time-dependence pattern elaborated in the paper.

There are some purely empirical explanations, however. One is captured by the econometric saying that volatility cannot be directly observed, which is used to emphasize that aggregation and sampling frequencies hide the more nuanced variability [16]. For instance, this paper used daily sampling frequency, but direct observability would require also assessments over the intra-day variability. Such short-run variability has been well-recognized in the software evolution and engineering research, as demonstrated by the different sliding window and burst models used in the mining of software repositories. This short-run development viewpoint is a typical candidate for further time series volatility modeling. Since the empirical experiment revealed clear daily volatility characteristics also in long-run, decade-wide software evolution time series, it seems reasonable to recommend that formal software evolution modeling should also ascertain that the fitted models are free of volatility effects. This can be done with the elaborated techniques.

The second conclusion relates to the dataset. Since volatility characteristics are present in the FreeBSD bug tracking system as well as mailing lists, it seems reasonable to conclude that churn as a concept is not related only to code. This is the same conclusion that has been previously investigated under a label of interactive socio-technical churn [10]. Not only does code churn increase the probability of mistakes, but the same likely applies to increasing variability in the technical and social software development environment. An interesting modeling context relates to the multivariate volatility models; does increasing volatility in one development system cause time-dependent variability in another system? An analogous further question relates to the surrounding environment; increasing variability in the environment may well translate to software development volatility. Thus, in general, perhaps a more meaningful volatility modeling context relates to the so-called business analytics.

The third conclusion is unambiguous: volatility did not increase in the proximity of the six major FreeBSD releases between 1995 and 2011. Although no direct comparisons can be made, this is a negative result with respect to the existing general empirical interpretations about the intensity near releases [7, 26]. Needless to say, further replicative empirical research would be required to evaluate the reason for this contrary result, or to contemplate whether the result is specific only to the FreeBSD dataset. There is also a twofold confirmation bias present: not only was a predefined set of features selected in advance, but systematic assessments were also omitted regarding the effect of milestone releases upon the conditional mean. The latter point restates the theoretical problems in applied volatility modeling: it is not always clear whether different events or shocks should be modeled in terms of the conditional mean, the conditional variance, or possibly even both.

This negative result does not invalidate the use of volatility modeling in different release engineering scenarios, however. As the paper demonstrates, volatility affects the predictive performance of longitudinal statistical models. If concepts such as time-to-release or time-to-market are considered, it is clear that the timings should not occur during volatility periods as these imply increasing uncertainty. A hectic development period is likely not the time to make judgments about the final software products.

To summarize, volatility modeling is plausible in software evolution research (P_1), although the empirical experiment shows no evidence that volatility would increase in the proximity of new releases (P_2). The likely most challenging further question relates to the theoretical meaning of volatility in software evolution and engineering time series.

5. REFERENCES

[1] T. Bollerslev, R. Y. Chou, and K. F. Kroner, "ARCH Modeling in Finance: A Review of the Theory and Empirical Evidence," *Journal of Econometrics*, vol. 52, no. 1–2, pp. 5–59, 1992.

[2] A. McWilliams and D. Siegel, "Event Studies in Management Research: Theoretical and Empirical Issues," *Academy of Management Journal*, vol. 40, no. 3, pp. 626–657, 1997.

[3] E. J. Barry, C. F. Kemerer, and S. A. Slaughter, "On the Uniformity of Software Evolution Patterns," in *Proceedings of the International Conference on Software Engineering (ICSE 2003)*. Portland: IEEE, 2003, pp. 106–113.

[4] R. S. Debreceny and G. L. Gray, "Data Mining of Electronic Mail and Auditing: Research Agenda," *Journal of Information Systems*, vol. 25, no. 2, pp. 195–226, 2011.

[5] S. A. Ross, "Information and Volatility: The No-Arbitrage Martingale Approach to Timing and Resolution Irrelevancy," *The Journal of Finance*, vol. 44, no. 1, pp. 1–17, 1989.

[6] I. Herraiz, J. M. González-Barahona, G. Robles, and D. M. Germán, "On the Prediction of the Evolution of Libre Software Projects," in *Proceedings of the International Conference on Software Maintenance (ICSM 2007)*. Paris: IEEE, 2007, pp. 405–414.

[7] D. Pagano and W. Maalej, "How Do Open Source Communities Blog?" *Empirical Software Engineering*, vol. 18, no. 6, pp. 1090–1124, 2013.

[8] K. Mohan and B. Ramesh, "Change Management Patterns in Software Product Lines," *Communications of the ACM*, vol. 49, no. 12, pp. 68–72, 2006.

[9] R. M. Bell, T. J. Ostrand, and E. Weyuker, "Does Measuring Code Change Improve Fault Prediction?" in *Proceedings of the 7th International Conference on Predictive Models in Software Engineering (Promise 2011)*. Banff: ACM, 2011, pp. 2:1–2:8.

[10] A. Meneely and O. Williams, "Interactive Churn Metrics: Socio-Technical Variants of Code Churn," *ACM SIGSOFT Software Engineering Notes*, vol. 37, no. 6, pp. 1–6, 2012.

[11] Y. Shin, A. Meneely, and L. Williams, "Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities," *IEEE Transactions on Software Engineering*, vol. 37, no. 6, pp. 772–787, 2011.

[12] M. M. Lehman, D. E. Perry, and J. F. Ramil, "On Evidence Supporting the FEAST Hypothesis and the Laws of Software Evolution," in *Proceedings of the Fifth International Software Metrics Symposium (METRICS 1998)*. Bethesda: IEEE, 1998, pp. 84–88.

[13] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller, "Mining Version Histories to Guide Software Changes," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 429–445, 2005.

[14] M. Goulão, N. Fonte, M. Wermelinger, and F. B. e Abreu, "Software Evolution Prediction Using Seasonal Time Analysis: A Comparative Study," in *Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR 2012)*. Szeged: IEEE, 2012, pp. 213–222.

[15] R. F. Engle, "Autoregressive Conditional Heteroskedasticity with Estimates of the Variance of United Kingdom Inflation," *Econometrica*, vol. 50, no. 4, pp. 987–1007, 1982.

[16] R. S. Tsay, *Analysis of Financial Time Series*. Chichester: John Wiley & Sons, 2002.

[17] A. K. Bera and M. L. Higgins, "ARCH Models: Properties, Estimation and Testing," *Journal of Economic Surveys*, vol. 7, no. 4, pp. 305–366, 1993.

[18] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, "Does Code Decay? Assessing the Evidence from Change Management Data," *IEEE Transactions on Software Engineering*, vol. 27, no. 1, pp. 1–12, 2001.

[19] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, A. Ihara, and K. Matsumoto, "The Impact of Mislabelling on the Performance and Interpretation of Defect Prediction Models," in *Proceedings of the 37th IEEE International Conference on Software Engineering (ICSE 2015)*. Florence: IEEE, 2015.

[20] M. Jørgensen and B. Kitchenham, "Interpretation Problems Related to the Use of Regression Models to Decide on Economy of Scale in Software Development," *Journal of Systems and Software*, vol. 85, no. 11, pp. 2494–2503, 2012.

[21] J. H. Stock, "Measuring Business Cycle Time," *Journal of Political Economy*, vol. 95, no. 6, pp. 1240–1261, 1987.

[22] N. Bettenburg and A. E. Hassan, "Studying the Impact of Social Interactions on Software Quality," *Empirical Software Engineering*, vol. 18, no. 2, pp. 375–431, 2013.

[23] C. Lokan and E. Mendes, "Investigated the Use of Duration-Based Moving Windows to Improve Software Effort Prediction: A Replicated Study," *Information and Software Technology*, vol. 56, no. 9, pp. 1063–1075, 2014.

[24] Y. Konchitchki and D. E. O'Leary, "Event Study Methodologies in Information Systems Research," *International Journal of Accounting Information Systems*, vol. 12, no. 2, pp. 99–115, 2011.

[25] J. Kleinberg, "Bursty and Hierarchical Structure in Streams," *Data Mining and Knowledge Discovery*, vol. 7, no. 4, pp. 373–397, 2003.

[26] S. Gala-Pérez, G. Robles, J. M. González-Barahona, and I. Herraiz, "Intensive Metrics for the Study of the Evolution of Open Source Projects: Case Studies from Apache Software Foundation Projects," in *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR 2013)*. San Francisco: IEEE, 2013, pp. 159–168.

[27] T. Bollerslev, "Generalized Autoregressive Conditional Heteroskedasticity," *Journal of Econometrics*, vol. 31, no. 3, pp. 307–327, 1986.

[28] A. C. MacKinlay, "Event Studies in Economics and Finance," *Journal of Economic Literature*, vol. 35, no. 1, pp. 13–39, 1997.

[29] S. Acharya, "Value of Latent Information: Alternative Event Study Methods," *The Journal of Finance*, vol. 48, no. 1, pp. 363–385, 1993.

[30] G. V. Henderson, Jr., "Problems and Solutions in Conducting Event Studies," *The Journal of Risk and Insurance*, vol. 57, no. 2, pp. 282–306, 1990.

[31] A. Ghalanos, "rugarch: Univariate GARCH Models. R Package Version 1.3-4," 2014, available online in May 2015: http://CRAN.R-project.org/package=rugarch.

[32] D. Spinellis, "A Tale of Four Kernels," in *Proceedings of the International Conference on Software Engineering (ICSE 2008)*. Lepzig: ACM, 2008, pp. 381–390.

[33] J. M. Osier, B. Kehoe, C. Support, and Y. Svendsen, "Keeping Track. Managing Messages with GNATS, the GNU Problem Report Management System," 2001, version 4.0-beta1. Available online in June 2014: http://www.gnu.org/software/gnats/doc/4_0_beta1/gnats.pdf.

[34] G. Kuk, "Strategic Interaction and Knowledge Sharing in the KDE Developer Mailing List," *Management Science*, vol. 52, no. 7, pp. 1031–1042, 2006.

[35] A. Silvestrini and D. Veredas, "Temporal Aggregation of Univariate and Multivariate Time Series Models: A Survey," *Journal of Economic Surveys*, vol. 22, no. 3, pp. 458–497, 2008.

[36] C. Izurieta and J. Bieman, "The Evolution of FreeBSD and Linux," in *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering (ISESE 2006)*. Rio de Janeiro: ACM, 2006, pp. 204–211.

[37] N. Jørgensen, "Developer Autonomy in the FreeBSD Open Source Project," *Journal of Management & Governance*, vol. 11, no. 2, pp. 119–128, 2007.

[38] R. F. Engle and T. Bollerslev, "Modelling the Persistence of Conditional Variances," *Econometric Reviews*, vol. 5, no. 1, pp. 1–50, 1986.

[39] D. Kwiatkowski, P. C. Phillips, P. Schmidt, and Y. Shin, "Testing the Null Hypothesis of Stationarity Against the Alternative of a Unit Root: How Sure Are We that Economic Time Series Have a Unit Root?" *Journal of Econometrics*, vol. 54, no. 3, pp. 159–178, 1992.

[40] B. Pfaff, *Analysis of Integrated and Cointegrated Time Series with R*, 2nd ed. New York: Springer-Verlag, 2008.

Estimating Product Evolution Graph using Kolmogorov Complexity

Yasuhiro Hayase
Division of Information Engineering
University of Tsukuba
Tsukuba, Japan
hayase@cs.tsukuba.ac.jp

Tetsuya Kanda
Graduate School of Information Science and Technology
Osaka University
Osaka, Japan
t-kanda@ist.osaka-u.ac.jp

Takashi Ishio
Graduate School of Information Science and Technology
Osaka University
Osaka, Japan
ishio@ist.osaka-u.ac.jp

ABSTRACT

This paper proposes a method of estimating a product evolution graph based on Kolmogorov complexity. The method *EEGL* applies lossless compression to the source code of products, then, presumes a derivation relationship between two products when the increase of information between the two products is small. An evaluation experiment confirms that EEGL and an existing method *PRET* tends to produce different errors when estimating evolution graph results.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering, Version control*; D.2.9 [**Software Engineering**]: Management—*Software configuration management*

General Terms

Algorithms

Keywords

Software evolution, evolution graph, estimation, Kolmogorov complexity, lossless compression

1. INTRODUCTION

When developing a new software product, an existing similar software product can be used as a *base product* – namely, a base product is copied and modified into the new product (i.e. *derived product*). Since a derived product shares source code with a base product, the derivation relationship should be recorded and managed in a consistent manner for the purpose of maintenance or aggregation of shared code. However, records of derivation are sometimes lost thorough software evolution process.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

IWPSE'15, August 30, 2015, Bergamo, Italy
ACM. 978-1-4503-3816-5/15/08...$15.00
http://dx.doi.org/10.1145/2804360.2804368

Kanda *et al.* [11] proposed *PRET*, a method to estimate derivation relationships between multiple software products (i.e. evolution graph). PRET employs an original distance measure for two software products that counts pairs of two similar source files whose longest common subsequence (LCS) of tokens is longer than certain ratio, across the two products. PRET builds a spanning tree of products based on number of similar files between products. Then, direction of an edge that means direction of derivation is estimated using the number of the tokens not included in the LCS.

Although PRET achieved high accuracy of estimation, there is still a room for improvement on the following points:

(1) Only tokens in source files are used for calculating product similarity. In other words, neither comments in source files nor other type of files, such as manuals or build scripts, do not contribute to the similarity. These documents also evolve in a similar manner as source files. Therefore, the documents can be used for estimating product derivation.

(2) File similarity is based on LCS of tokens. In the case of a code fragment is moved, LCS does not include tokens in the moved fragments, then file similarity is get lower.

(3) Only number of tokens impact file similarity. Therefore, newness and complexity of tokens do not impact the similarity. For example, a completely new and unique code fragment and a code fragment made by copy-and-paste (code clone) have same impact. Likewise, a frequently appearing reserved word and a first appearance of long and complex string literal have same impact.

(4) If a source file is split into two segment files, both of the segment files are judged dissimilar to the origin file. Since the origin file have many tokens that are not included in each segmented file. Merging of source files involves same problem.

(5) (minor problem) To obtain tokens from source files, PRET need to recognize a programming language used for each source file.

To deal with the above issues, this paper proposes a new method **EEGL** (Estimating Evolution Graph using Lossless compression) to estimate a product evolution graph using increase of information in the sense of Kolmogorov complexity [20, 21, 13, 3]. Kolmogorov complexity is a complexity measure for a string; it is defined as the length of the

shortest program that output the string. Since Kolmogorov complexity is not computable for arbitrary strings, lossless compression methods are used as upper bound (e.g. approximate) function for Kolmogorov complexity. [5, 16, 12, 7] In our approach, divergence from product p to product q is measured as difference between the size of a compressed archive of p and a compressed archive of p with q.

EEGL solves the issues on PRET: 1) documents, Makefile, and comments in a source file is equally used for divergence calculation. 2) When a code fragment is moved, size of compressed archive gets slightly bigger. However, the increase must be small compare to moved fragment. 3) Since code clones are redundant, increase of compressed archive is small if a big code clone is added. New appearance of long complex string literal brings deserved increases to a compressed archive. 4) File split or merging without changing total contents has small impact to compressed size. 5) Proposed approach is language independent.

The rest of this paper is organized as follows. Section 2 describes detailed idea and procedure of our approach. In section 3, preliminary investigation of lossless compression tools is performed. Section 4 shows comparison with PRET using same data sets of evaluation experiment in [11]. Section 5 shows related work. Finally, conclusion and future work are presented in Section 6.

2. APPROACH

Proposed approach EEGL estimates evolution graph by comparing the compressed size of a single product and compressed size of merging of two different products. The approach is based on two basic ideas. First idea is that pair of products that are in direct derivation relationship should share more information than the pairs that are not in direct derivation relationship. This fact is confirmed by Arbuckle[1]. Second idea is that a derived product tends to have more quantity of information than a base product. It is known that values of size or complexity metrics for sequential releases have strong tendency to increase. [14, 15]

Procedure to estimate the evolution graph is described below:

Input Source code set of n products $P = \{p_1, p_2, ...p_n\}$ (p_i is source code of i-th product)

Output Directed Graph $G = (P, E)$. Edge set E means estimation of a product derivation relationship. Node set P is identical to input.

Step 1. Remove binary files (e.g. pictures) from each of $p_i (i = 1..n)$.

Step 2. For each $p_i (i = 1..n)$, make `tar` archive in order of path name, and then apply lossless compression to the tar archives. Define $Z(p_i)$ as compressed data size of p_i.

Step 3. For all pair of products (p_i, p_j) where $p_i, p_j \in P$, make a combined tar ball $p_i \cdot p_j$. Combined tar ball composed from all source files in both p_i and p_j. Order of files in tar ball is ascendant order of the path name except for top directory name. Example of file order is shown in table 1. This order is expected to help lossless compression algorithms to find similar parts in files, since files that have similar path names tend to have similar contents.

Table 1: Example of file orders in single and combined products

a	`a/0/1.c a/1.c a/4.c`
b	`b/0/2.c b/1.c b/3.c`
$a \cdot b$	`a/0/1.c b/0/2.c a/1.c b/1.c b/3.c a/4.c`

Step 4. Compress all combined tar balls. Define $Z(p_i \cdot p_j)$ as compressed data size of $p_i \cdot p_j$.

Step 5. Finally, make product derivation edges E based on $I(p_i, p_j) = Z(p_i \cdot p_j) - Z(p_i)$, i.e. *increase of information from p_i to p_j*, and following assumptions.

ASM1. Quantity of information of a derived product is larger than base product.

ASM2. Increase of information from true base product is small, compared to increase from other products such as ancestor or sibling products.

ASM3. Number of base product of each product is at most one.

Based on these assumptions, directed edge set E of estimated evolution graph $G = (P, E)$ is defined as follows.

$$(p, q) \in E \iff Z(p) < Z(q) \;\wedge\; (\forall r \in P, \quad Z(r) \geq Z(q) \;\vee\; I(r, q) \geq I(p, q))$$

This definition means that for all product q, make a edge from p to q when increase of information from p is smallest in the products whose compressed size is smaller than q.

3. PRELIMINARY INVESTIGATION: COMPARISON OF COMPRESSION TOOLS

To evaluate aptitude of lossless compression algorithms for estimating derivation relationships, we performed preliminary investigation of examining divergence of program versions and increase of compressed data sizes. Specifically, we compressed the all possible combined products of two different versions by several compression tools, and then observed compressed data size. Following three compression tools are compared: gzip (deflate (LZ77[23] + Huffman coding[9])), bzip2 (block sorting [2] + Huffman coding), xz (LZMA (Variant of LZ77 + Range Encoder[18])). All tools are executed with `-9` option, which controls compression ratio. Target software products and versions are 8.0.0 to 8.0.26 (totally 27 releases) of PostgreSQL[1] source code.

First of all, summary compression ratio of each single release is shown in Table 2. Order of average compression ratio is gzip > bzip2 > xz. This result confirms that newer compression tools achieve better compression ratio. On the other hand, this result also shows compression ratio of bzip2 is unstable, since stddev compare to average of compression ratio is bigger than others.

Next, we observed $\frac{Z(p \cdot q) - Z(p)}{Z(p)}$, that is to say, relative increase ratio of compressed data size when release q is added

[1] `http://www.postgresql.org/`

Table 2: Compression ratio for each single release

Tool	Compression Ratio (average ± stddev)	stddev/average
gzip -9	0.231 ± 0.000384	0.00166
bzip2 -9	0.182 ± 0.000368	0.00202
xz -9	0.163 ± 0.000233	0.00143

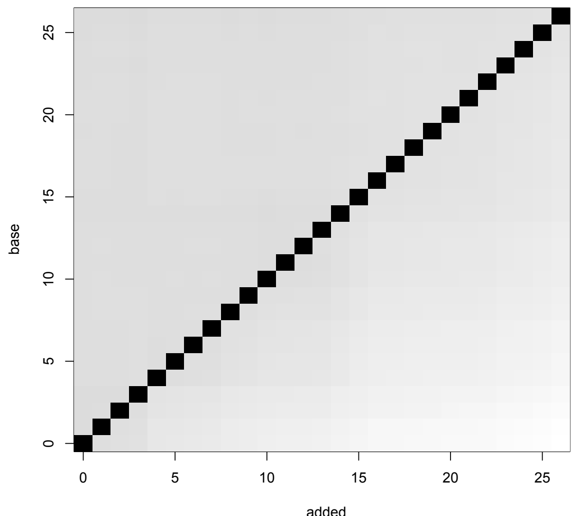


Figure 1: Heatmap of relative increase for gzip -9

to release p. Figure 1 to 4 are a heat map whose cells express the relative increase ratio. Vertical axis means minor release number (x of $8.0.x$) of base product p. Horizontal axis means minor release number of added product q. Darker cells means lower relative increase; brighter cells means higher relative increase. Diagonal cells in the heatmaps is black that means very low (almost zeor) increase ratio, since the diagonal cells correspond to the case that two identical products are combined. Figure 1, 2, and 3 are cases of gzip, bzip2, and xz respectively, and employs 0.5 for gray gamma curve. Figure 4 is also the same case of gzip except that gamma value is 0.1, because Figure 1 is difficult to read subtle differences between cells.

If the heat map of a compression tool monotonically gets brighter from diagonal cells to right, that means the compression tool accurately catches increase of information. This is because difference of information contained in the source code of two releases should be bigger if the distance of the releases is far.

Let us consider the monotonicity of the three tools. Total brightness of Figure 1 means difference of relative increase ratio of gzip is smaller than other tools. However, we can see that monotonicity of gzip is fine from Figure 4, even though the difference is small. Dappling in Figure 2 reveals anti-monotonicity of bzip2. Figure 3 shows the best monotonicity of xz.

From the above observation, xz is expected to output the most accurate evolution graph, followed in order by gzip and bzip2.

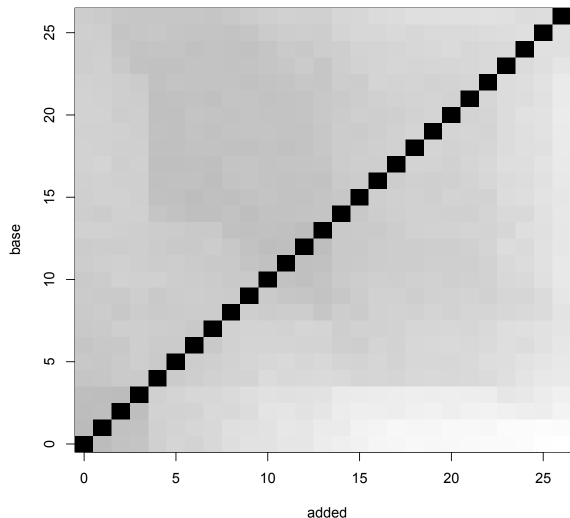


Figure 2: Heatmap of relative increase for bzip2 -9

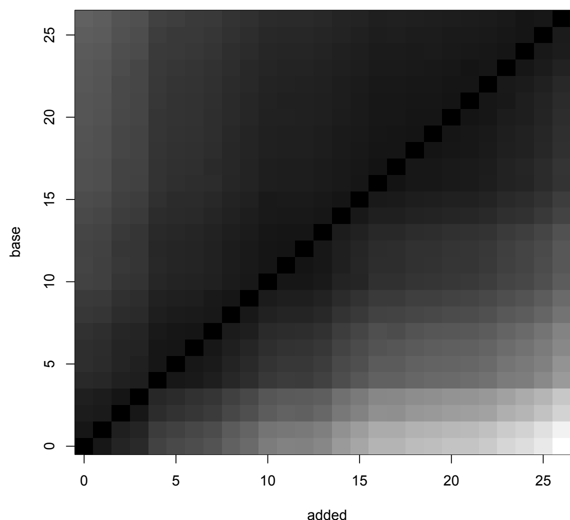


Figure 3: Heatmap of relative increase for xz -9

4. EVALUATION EXPERIMENT: COMPARISON WITH PRET

This section describes the comparison of EEGL with existing approach PRET, using the same data sets used to evaluated PRET. (data sets are available on our website [2]) Evaluation basis is accuracy of estimated evolution graphs. On executing PRET, its threshold parameter of file similarity is set to 0.9 where the best accuracy was achieved in the data sets. All the target projects are implemented in C.

Six data sets are designed to cover the following situation.

[2] `http://sel.ist.osaka-u.ac.jp/pret/`

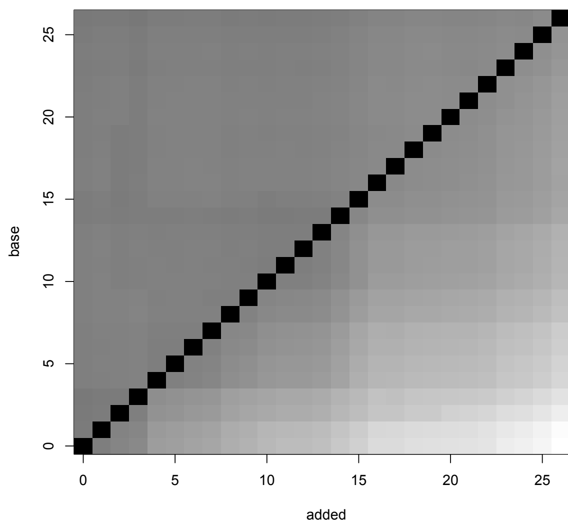


Figure 4: Heatmap of relative increase for gzip -9 (Gamma=0.1)

- Single project
  1. a simple straightforward evolution
  2. an entire history with branching and parallel evolution
  3. a limited history to recent releases, and
  4. a limited history that intermediate variants are missing
- Forked projects
  5. a history with two forked projects
  6. a complex history with more than two projects including branching and merging

To evaluate the accuracy for analyzing a single project, we arranged four data sets by selecting releases from PostgreSQL database project. They have a branch of major releases and branches for maintaining each minor releases.

dataset1 Pgsql-major: 12 major releases of PostgreSQL from 7.0 to 9.2.0. This dataset emulates sequential evolution of software product.

dataset2 Pgsql-8-ALL: all minor releases in PostgreSQL version 8. This dataset contains several branching and parallel evolutions with large number of variants.

dataset3 Pgsql8-latest: take up to 5 last minor releases from each major branches of PostgreSQL 8. This dataset emulates the situation that old releases in branches are lost.

dataset4 Pgsql8-annually: 25 versions of PostgreSQL. This dataset emulates the situation that developers have an incomplete set of variants. The latest variant in our dataset is released in September 2012 so we picked up the variants released around every September from 2005 to 2012.

To evaluate the accuracy for analyzing variants with project fork, we selected *FFmpeg*[3] and *Libav*[4] project which are libraries for processing multimedia data. They were originally a single project, but Libav is forked from FFmpeg with developers.

dataset5 FFmpeg: Several adjacent releases around project fork into FFmpeg and Libav.

We also selected BSD family operating systems: 4.4BSD, FreeBSD[5], NetBSD[6], and OpenBSD[7]. Those operating systems are derived from 4.3BSD but now they are independent projects. They affected each other after branching so we can say that those projects have complex evolution relationships with branching and merging.

dataset6 BSD: releases of 4.4BSD, FreeBSD, NetBSD, and OpenBSD. These products are in complex evolution relationships including branching and merging.

Table 3 shows summary of evaluation result. Column *#true-edges* shows number of edges in the true evolution graph. Column *#output* shows the number of edges in an estimated graph output by a tool. Column *#correct* shows number of correct edges, i.e. the edges included in both the estimation output and the true evolution graph. Column *precision* and *recall* shows ratio of #correct divided by #output and #true-edges respectively. In each dataset, the best values of precision and recall are emphasized by bold face and underline. Column set *#errors* provides breakdown of error reason. Column *reverse* shows the number of error edges whose reverse edges are include in the true evolution graph. Column *skip* shows the number of error edges that are not come from the direct base product, but from upstream of the direct base product. For example, 2-skip error means an edge is come from *true base product of base product of base product.*

Only for dataset1, dataset5 using gzip, and dataset6 using gzip or bzip2, EEGL shows better accuracy than PRET. For dataset4, EEGL ties with PRET only when gzip or xz is used. For dataset2 and dataset3, accuracy of EEGL is worse than PRET.

Figure 5 shows example of estmated graphs for dataset5 obtained by EEGL with gzip and PRET. Thin black arrows are correct edges. Incorrect edges are shown as colored bold edges with reason. Dashed gray arrows are correct edges but not include in the estimation. We can see that direction of derivation is correctly estimated by EEGL. In both graphs, branching after FFmpeg 0.5.4 is misaligned to Libav or FFmpeg 0.5.5.

4.1 Discussion on Evaluation Result

EEGL achieved higher precision on dataset1, and comparable precision on dataset4, therefore EEGL could be good at large change between products. When a product faces

[3] `http://www.ffmpeg.org/`
[4] `http://libav.org/`
[5] `http://www.freebsd.org/`
[6] `http://www.netbsd.org/`
[7] `http://www.openbsd.org/`

Table 3: Evaluation Result

data	#true-edges	method	#output	#correct	precision	recall	#errors			
							reverse	skip		
								1	2	≥ 3
dataset1	12	EEGL(gzip)	12	12	**1.00**	**1.00**	0	0	0	0
		EEGL(bzip2)	12	12	**1.00**	**1.00**	0	0	0	0
		EEGL(xz)	12	12	**1.00**	**1.00**	0	0	0	0
		PRET	12	11	0.917	0.917	1	0	NA	NA
dataset2	143	EEGL(gzip)	143	105	0.734	0.734	3	22	5	3
		EEGL(bzip2)	143	58	0.406	0.406	11	22	8	33
		EEGL(xz)	143	122	0.853	0.853	4	9	2	0
		PRET	143	130	**0.909**	**0.909**	8	1	NA	NA
dataset3	37	EEGL(gzip)	37	24	0.649	0.649	0	6	1	0
		EEGL(bzip2)	37	21	0.568	0.568	0	4	4	1
		EEGL(xz)	37	29	0.784	0.784	0	1	1	0
		PRET	37	32	**0.865**	**0.865**	0	0	NA	NA
dataset4	24	EEGL(gzip)	24	20	**0.833**	**0.833**	0	0	0	0
		EEGL(bzip2)	24	14	0.583	0.583	0	2	3	1
		EEGL(xz)	24	20	**0.833**	**0.833**	0	0	0	0
		PRET	24	20	**0.833**	**0.833**	0	0	NA	NA
dataset5	15	EEGL(gzip)	15	14	**0.933**	**0.933**	0	0	0	0
		EEGL(bzip2)	15	1	0.0667	0.0667	2	4	1	1
		EEGL(xz)	15	7	0.467	0.467	3	4	0	0
		PRET	15	11	0.733	0.733	2	1	NA	NA
dataset6	17	EEGL(gzip)	15	10	**0.667**	**0.588**	0	0	0	0
		EEGL(bzip2)	15	10	**0.667**	**0.588**	0	0	0	0
		EEGL(xz)	15	8	0.533	0.471	1	1	0	0
		PRET	15	9	0.600	0.529	3	0	NA	NA

large change, source files may be split or merged, or large part of a source file may be rewritten. Such cases delude PRET, since PRET stands on finding similar files with certain similarity threshold. Since EEGL does not employ any threshold and compression tools are able to find similar portion in considerably modified files, EEGL may have a capability to evaluate similarity of products that faces large change correctly.

In case that bzip2 is used in EEGL, accuracy is totally lower than others. Especially, number of skip-errors is outstanding in the results. Considering both this result and anti-monotonicity in the preliminary investigation, bzip2 is possibly not suitable for approximating Kolmogorov complexity of source code.

On the other hand, contrary to the expectation from the preliminary result, not only xz and gzip ties on some data sets, but also precision of xz is far lower than gzip for dataset5. To clarify the difference between gzip and xz, additional experiment with more data sets is required.

One superior point of EEGL to PRET is parameter-free. EEGL only requires product set and compression tools. PRET requires threshold of file similarity as a parameter on execution. This parameter substantially affects estimation result. Therefore, execution of EEGL is not optimized for the data sets.

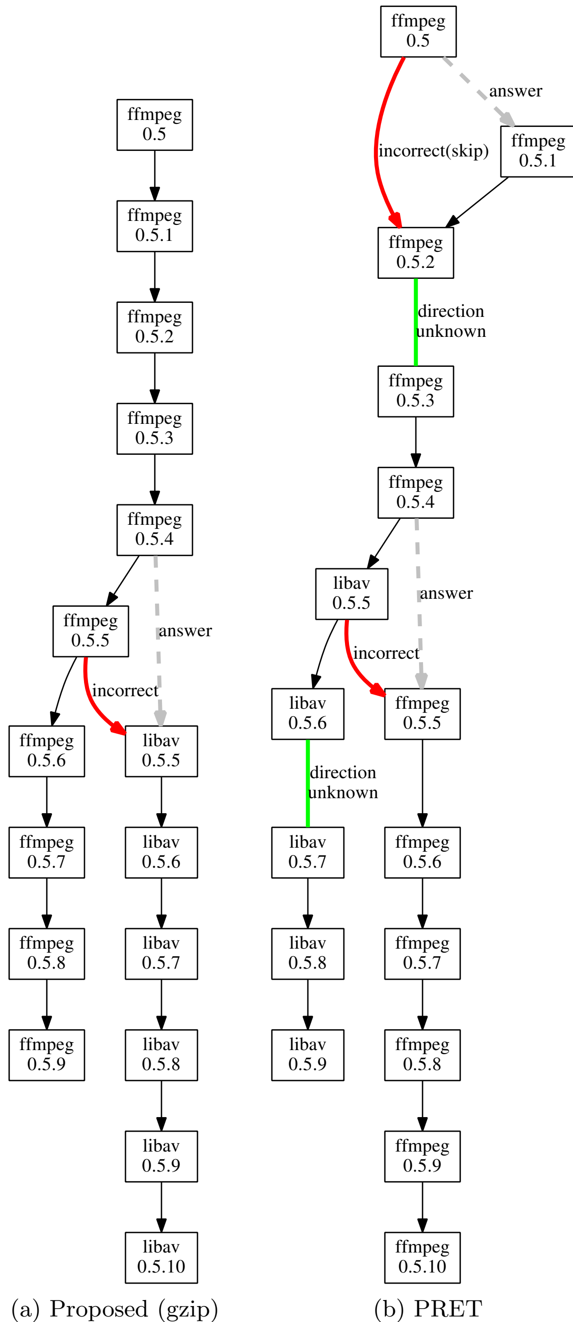
Through observation of breakdown of the errors, EEGL tends to output more skip errors and less reverse errors in comparison with PRET. The different tendency of errors indicates the chance of improving accuracy by combining the two approaches. In addition, number of skip errors varies according to compression tool or data set. This variation also indicates that combination of compression tools have potential to reduce skip errors.

5. RELATED WORK

Arbuckle [1] proposed to use normalized compression distance (NCD) [5], which is based on Kolmogorov complexity, between versions of a software product for understanding software evolution. NCD is used as measures of quantity of information shared between the versions. He employed gzip(zlib), bzip2(bzlib), ppmd, and lzma as compression algorithms for NCD calculation, then compred the algorithms. Through the comparision, gzip could not catch the difference of versions except for consecutive and nearly identical versions. Bzip2 shows instability of compression ratio. Due to this, gzip and bzip2 were judged unsuitable for understanding software evolution. In contrast, gzip worked fine in our experiment. This difference should result from difference of the objective of the two approaches because finding a consecutive version is enough for EEGL.

Kirk and Jenkins [12] used Kolmogorov complexity for evaluating software obfuscation. They employed bzip2 for approximation of Kolmogorov complexity.

For the purpose of evaluating the value of software product or services, Fujiwara *et al.* [7] proposed a method to analyze the result of interview or questionnaire for stakeholders using Kolmogorov complexity. Result of interview or questionnaire are hierarchically clustered using distance measure based on Kolmogorov complexity. In this study, gzip is used for approximating Kolmogorov complexity.



(a) Proposed (gzip) (b) PRET

Figure 5: Estimated graphs for dataset5

Edit distance is often used for comparing program versions. For example, diff command[8] finds difference between two files. Diff treats each file as a sequence of text lines, and then calculates the shortest edit script which consists of two kind of basic edit operations, insertion and deletion of a line, between the two sequences. [10, 17, 19, 22] Difference computation between tree structure [4], which allows insertion, deletion, and update of node and moving of sub-tree as basic operation, is also used for comparing source code. [6, 8] The increase of information $I(p_i, p_j) = Z(p_i \cdot p_j) - Z(p_i)$ in this paper can be interpreted as edit distance from product p to q when internal actions of a compression/reconstruction tool are permitted as basic edit operations.

[8] `http://www.gnu.org/software/diffutils/`

6. CONCLUSION AND FUTURE WORK

This paper proposes a method to estimate a product evolution graph using Kolmogorov complexity. Result of evaluation experiment shows the difference in errors with existing method PRET. Lossless compression tools used for approximating Kolmogorov complexity also affect estimation result.

We are planning to improve estimation accuracy by combination with PRET, combination of lossless combination tools, or adopting another lossless compression algorithms.

7. ACKNOWLEDGEMENT

This work was supported by KAKENHI 25730036 and KAKENHI 25220003.

8. REFERENCES

[1] T. Arbuckle. Studying software evolution using artefacts' shared information content. *Sci. Comput. Program.*, 76(12):1078–1097, Dec. 2011.

[2] M. Burrows, D. J. Wheeler, M. Burrows, and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, 1994.

[3] G. J. Chaitin. On the length of programs for computing finite binary sequences. *J. ACM*, 13(4):547–569, Oct. 1966.

[4] S. S. Chawathe, A. Rajaraman, and H. G.-M. an d Jennifer Widom. Change detection in hierarchically structured information. *SIGMOD Rec.*, 25(2):493–504, 1996.

[5] R. Cilibrasi and P. Vitanyi. Clustering by compression. *Information Theory, IEEE Transactions on*, 51(4):1523–1545, April 2005.

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

[7] Y. Fujiwara, T. Gotoh, and H. Iguchi. Product/service value validation based on kolmogorov complexity (in Japanese). In *Proceedings of Forum on Information Technology 2009*, volume 8, pages 55–62. FIT committee, August 2009.

[8] Y. Hayase, M. Matsushita, and K. Inoue. Revision control system using delta script of syntax tree. In *Proceedings of the 12th International Workshop on Software Configuration Management*, SCM '05, pages 133–149, New York, NY, USA, 2005. ACM.

[9] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40(9):1098–1101, September 1952.

[10] J. W. Hunt and M. D. McIlroy. An algorithm for differential file comparison. Technical Report CSTR 41, Bell Laboratories, Murray Hill, NJ, 1976.

[11] T. Kanda, T. Ishio, and K. Inoue. Extraction of product evolution tree from source code of product variants. In *Proceedings of the 17th International Software Product Line Conference*, SPLC '13, pages 141–150, 2013.

[12] S. R. Kirk and S. Jenkins. Information theory-based software metrics and obfuscation. *Journal of Systems and Software*, 72(2):179 – 186, 2004.

[13] A. N. Kolmogorov. Three approaches to the quantitative definition of information. *International Journal of Computer Mathematics*, 2(1-4):157–168, 1968.

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

[15] M. Lehman, J. Ramil, P. Wernick, D. Perry, and W. Turski. Metrics and laws of software evolution-the nineties view. In *Software Metrics Symposium, 1997. Proceedings., Fourth International*, pages 20–32, Nov 1997.

[16] M. Li, X. Chen, X. Li, B. Ma, and P. Vitanyi. The similarity metric. *Information Theory, IEEE Transactions on*, 50(12):3250–3264, Dec 2004.

[17] D. Maier. The complexity of some problems on subsequences and supersequences. *J. ACM*, 25(2):322–336, Apr. 1978.

[18] G. N. N. Martin. Range encoding: an algorithm for removing redundancy from a digitized message. In *Proceedings of the Video & Data Recording Conference*, Southampton, Jul. 1979.

[19] E. W. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1:251–266, 1986.

[20] R. Solomonoff. A formal theory of inductive inference. part I. *Information and Control*, 7(1):1 – 22, 1964.

[21] R. Solomonoff. A formal theory of inductive inference. part II. *Information and Control*, 7(2):224 – 254, 1964.

[22] S. Wu, U. Manber, G. Myers, and W. Miller. An O(NP) sequence comparison algorithm. *Inf. Process. Lett.*, 35(6):317–323, 1990.

[23] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theor.*, 23(3):337–343, Sept. 2006.

Using Control Flow Analysis to Improve the Effectiveness of Incremental Mutation Testing*

Luke Bajada
PEST Research Lab,
University of Malta
luke.bajada@um.edu.mt

Mark Micallef
PEST Research Lab,
University of Malta
mark.micallef@um.edu.mt

Christian Colombo
PEST Research Lab,
University of Malta
christian.colombo@um.edu.mt

ABSTRACT

Incremental Mutation Testing attempts to make mutation testing less expensive by applying it incrementally to a system as it evolves. This approach fits current trends of iterative software development with the main idea being that by carrying out mutation analysis in frequent bite-sized chunks focused on areas of the code which have changed, one can build confidence in the adequacy of a test suite incrementally. Yet this depends on how precisely one can characterise the effects of a change to a program. The original technique uses a naïve approach whereby changes are characterised only by syntactic changes. In this paper we propose bolstering incremental mutation testing by using control flow analysis to identify semantic repercussions which a syntactic change will have on a system. Our initial results based on two case studies demonstrate that numerous relevant mutants which would have otherwise not been considered using the naïve approach, are now being generated. However, the cost of identifying these mutants is significant when compared to the naïve approach, although it remains advantageous when compared to traditional mutation testing so long as the increment is sufficiently small.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Reliability, Performance

Keywords

Test Suite Adequacy Analysis, Incremental Mutation Testing, Dataflow Analysis

*Project GOMTA financed by the Malta Council for Science & Technology through the National Research & Innovation Programme 2013

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

IWPSE'15, August 30, 2015, Bergamo, Italy
ACM. 978-1-4503-3816-5/15/08...$15.00
http://dx.doi.org/10.1145/2804360.2804369

1. INTRODUCTION

Mutation testing [9] is a technique which analyses the adequacy of a test suite using fault injection. Whilst it has been shown to be more effective in finding test suite deficiencies than other measures such as code coverage analysis, the expense associated with the approach is still a significant barrier to entry to the industry. To improve the performance, we have proposed a way of splitting the cost of mutation testing over the iterations of the software development process [4]. This was achieved by applying mutation testing to only the changed parts of the code base since a previous commit, thus carrying out mutation testing incrementally over time.

The main challenge in this approach is to precisely characterise the semantic impact of a syntactic change in the code base, given that a change in one part of a program can affect parts which would have been modified indirectly through calls to changed methods, data flows, or through shared resources with modified parts of the system.

In this paper, we recount our experience of attempting to improve the technique by leveraging control flow analysis (see Section 3). In particular, the paper discusses work done to answer the following research questions:

RQ1: Can the effectiveness of incremental testing be improved by using control flow analysis to more precisely characterise the effects of changes between two evolutions of a program?

RQ2: What tradeoffs exist between expense and effectiveness when using this approach?

Two case studies were carried out as part of our research (see Section 4), one on an open source project and the other on an industry case study with a partner in the payment processing industry.

2. BACKGROUND

In this section, we provide a brief overview of incremental mutation testing and relevant static analysis techniques which were used in this research.

2.1 Incremental Mutation Testing

In essence, mutation testing works as follows: given a program P and a test suite T which tests P, the approach involves generating faulty variations of P (called mutants) and checking whether for every mutant, there is at least one test case in T which fails. We write $T(P)$ to denote a successful run of test suite T on program P

and $\neg T(P)$ to denote that at least one of the tests in the test suite has failed on P.

Mutation testing begins by generating a set of programs $P_1, P_2, \ldots, P_n$ using a set of mutation operators represented by the function $\mathcal{M}$ on the program P, $\mathcal{M}(P) = \{P_1, P_2, \ldots, P_n\}$. These programs are usually syntactically similar to P but never (syntactically) equivalent to it. That is to say $\forall i : 1..n \cdot P_i \not\equiv P$.

Subsequently, T is executed against all $P_i \in \mathcal{M}(P)$. For each mutant P_i, if at least one test results in a failure, we say that T has *killed* the mutant. Otherwise, we say that the mutant remains *unkilled*. Unkilled mutants might indicate a diminished adequacy of T with respect to P. This brings us to our definition of coverage.

Definition 1. (Coverage) A test suite T is said to cover a program P, denoted $T \triangleright P$ if and only if P satisfies T, $T(P)$, while any $P_i \in \mathcal{M}(P)$ fails the test suite, $\neg T(P_i)$:

$$T \triangleright P \stackrel{\text{def}}{=} T(P) \wedge \forall P_i \in \mathcal{M}(P) \cdot \neg T(P_i)$$

The ratio of killed mutants to total mutants is known as the mutation score and provides a measure of test suite coverage in the context of the generated mutants. Mutation operators are usually designed to change P in a way that corresponds to a fault which could be introduced by a developer. Consequently, in comparison to techniques such as statement coverage analysis, mutation testing provides a significantly more reliable measure of test suite thoroughness [3, 5]. Despite its effectiveness, mutation testing suffers from a significant problem [9]: Whilst the polynomial computational complexity of mutation testing does not seem prohibitive, in a typical commercial system the large amount of potential mutation points would make the computational expense considerably high. Furthermore, once mutants have been generated, each one needs to be tested against the original program's test suite. Considering that test suites on large systems will optimistically take a few minutes to execute, the time required for this task would be considerable.

Incremental mutation testing attempts to alleviate the prohibitive computational expense associated with mutation testing by leveraging the evolutionary nature of modern software development practices such as Agile development. The underpinning idea is that of limiting the scope of mutation testing to code that has changed within the context of two particular versions of the code. This effectively selects areas of the code with elevated *code churn* — a measure of changes made to a software component over a period of time which has been shown to effectively predict defect density during system evolution [12]. By applying mutation testing on each change across successive versions of the code, over the entire evolutionary process, one would have effectively applied mutation testing over the whole system, incrementally. More precisely, incremental mutation testing assumes two programs P and P_{ev} where P_{ev} is an evolution of P such that P_{ev} consists of two parts: a changed part (P^{δ}_{ev}) which has evolved from a corresponding part of P (P^{δ}), and an unchanged part ($P^{\not\delta} = P^{\not\delta}_{ev} = P^{\not\delta}$) with respect to P. We therefore represent P and P_{ev} as $P = P^{\delta} + P^{\not\delta}$ and $P_{ev} = P^{\delta}_{ev} + P^{\not\delta}$. In this context, the composition operator + assumes that there is a way of splitting a program into two parts such that the parts can be tested independently. Similarly, the technique assumes that there is a way of splitting the test suite into (potentially overlapping) parts which test the corresponding program parts. Formally:

Definition 2. (Independent testability) For any program $P = P^{\delta} + P^{\not\delta}$ and test suite $T = T^{\delta} + T^{\not\delta}$, the program passes the test suite if and only if its parts pass the corresponding parts of the test suite respectively:

$$T(P) \iff T^{\delta}(P^{\delta}) \wedge T^{\not\delta}(P^{\not\delta})$$

Using these assumptions, given that a test suite has been shown to adequately cover a system under test, in the following evolution of the code, this information can be used to minimise the number of mutations required to check the test suite. Intuitively, this is achieved by eliminating the unchanged part of the system from mutation testing: if the second version of the code can be split into the changed part and the unchanged part, incremental mutation testing assumes that tests relating to the unchanged part do not need to be analysed for thoroughness because this would have been done in previous evolutions of the code. More formally, this idea is captured in the following theorem:

Theorem 1 (Incremental mutation testing). *If the system code $P = P^{\delta} + P^{\not\delta}$ has been shown to be adequately covered by a test suite T, $T \triangleright (P^{\delta} + P^{\not\delta})$, then to show that the new version is also adequately covered, $T_{ev} \triangleright (P^{\delta}_{ev} + P^{\not\delta})$, it suffices to check that $T^{\delta}_{ev} \triangleright P^{\delta}_{ev}$:*

$$T \triangleright (P^{\delta} + P^{\not\delta}) \wedge T^{\delta}_{ev} \triangleright P^{\delta}_{ev} \implies T_{ev} \triangleright (P^{\delta}_{ev} + P^{\not\delta})$$

Whilst this theorem has been shown to be true [4], the significant assumptions described above which make the theorem work, cannot be overlooked. In particular, being able to split a program into the changed and the unchanged part such that the subparts can be tested independently, is in general a difficult task. In our previous work [4], we have taken the naïve approach of simply separating the new/changed methods from the unchanged methods (and their corresponding tests respectively). Clearly, this approach does not satisfy our assumption since this does not guarantee independent testability — for example changed methods might be called from unchanged methods, or unchanged methods may share global variables with changed ones.

In order to mitigate this issue, we propose to use control flow analysis techniques to analyse the structure of the code under consideration and find better approximations of the impact of a syntactic change. This should enable us to move closer to satisfying the assumption which is crucial for Theorem 1 to hold.

2.2 Static Analysis

Due to the issue of independent testability (see Definition 2), we attempt to utilise information about the *coupling* within a software system — a qualitative measure that shows the degree of interdependence between modules. Whilst Lethbridge and Lagamiere [10] identify nine types of coupling, in this work we choose to target *routine call* coupling since the design principle of *separation of concerns* in the object-oriented paradigm naturally leads to a common occurrence of this type of coupling as objects delegate functionality to each other through method calls. To this extent, we turn our attention to static analysis which enables us to elicit the necessary information about the coupling features of interest.

Control flow analysis is used to determine the control flow relationships in a program [2]. The control flow is the order in which statements or function calls in code are executed. Of particular interest to us is Shivers' work in identifying the possible set of callers

to a function or method call [14]. This is because, due to the notion of indirect inputs[1], a change in a particular method is likely to affect the behaviour of methods which call (or transitively call) the changed method. Shivers' work uses a call graph as the internal representation of choice during static analysis. The call graph of a program is a directed graph that represents the relationships between the program's procedures or methods [6]. According to the method being analysed in the call graph, the calling relationships between that method and the methods that call it can span over several depths.

In the next section, we explain how using the call graph, we identify the indirectly affected parts of a code base so that this is considered for mutation testing.

3. PROPOSED APPROACH

Recall from Section 2.1 that in incremental mutation testing, we consider a program P_{ev}, an evolution of P, to be composed of two parts $P_{ev}^{\delta} + P^{\not\delta}$. In [4], we approximate P_{ev}^{δ} by identifying the set of methods (which we will refer to as M^{δ}) that contain syntactic differences between P and P_{ev}. We consider this approach as being naïve because the impact of a syntactic change to the code base is not necessarily limited to the location of that change — unchanged locations of the code which depend on changed locations could potentially be impacted. Therefore our naïve approach risks a situation whereby potentially valuable mutants would not be generated and analysed.

In this work, we improve the approximation of P_{ev}^{δ} by including a set of methods (referred to as *callSet*) which call (transitively up to a particular depth) the methods in M^{δ} as follows:

Definition 3. (callSet) Assuming we have a call graph $\langle V, E\rangle$ where V represents methods and E represents tuples of methods (m', m) signifying that m' calls m,

$$callSet(M^{\delta}, d) =$$

$$M^{\delta} \cup \begin{cases} \emptyset & \text{if } d = 0 \\ \bigcup_{m \in M^{\delta}} callSet(\{m' \in V \mid (m', m) \in E\}, d-1) & \text{otherwise} \end{cases}$$

Intuitively, consider a call graph of a program as depicted in Figure 1. The grey circle in the middle of the call graph represents a method in which a syntactic change has occurred whilst unshaded circles connected by a directed edge indicate pairs of methods in which one method calls the other (as indicated by the direction of the edge). Whereas in [4], the naïve approach only considered the shaded circle for analysis, in this paper we also take the other circles into account. We are only interested in including methods which call (or transitively call) a method where a syntactic change has occurred because their behaviour could potentially be effected by the change.

Note that we refrain from defining the value of d because as part of our investigation we look at how various depth values affect the performance of our approach.

[1]The notion of indirect inputs refers to situations whereby method behaviour is influenced by means other than parameter values, most commonly return values of called methods [11].

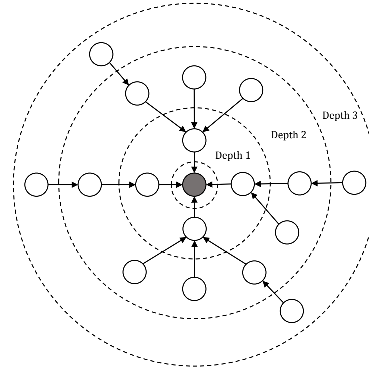


Figure 1: Example of call graph analysis at various depths

Finally, we remark that we do not extend this same approach to refine which tests are executed for each mutant, since in [4] we already selected tests which directly call mutated methods. We consider this to be adequate since we are dealing with unit tests (which should be only concerned with testing the called methods directly).

4. EVALUATION

As with most approximation techniques, there is a trade off between effectiveness and performance. In general, a more precise approximation comes with a higher cost. To this end, our evaluation takes the form of a cost-benefit analysis of the proposed technique via two case studies: one is an open source project having 5KLOC, which has also been used in our previous study [4] — the Apache Commons CLI library, while the second is an 10KLOC industrial system provided by a partner in the payments processing industry who utilises an iterative development cycle. The CLI library was selected in order to maintain consistency with our previous study, as well as having the characteristic of being very mature (thirteen years old with 97% code coverage) and maintained by a community of developers. On the other hand, the payments processing system was selected in order to provide a case study of a system still under development (one year old with 60% code coverage) by a focused team of developers.

Following our previous approach, we selected three different time periods in the life time of each project such that: (*i*) all periods had the same commencement timestamp and (*ii*) the end timestamp of each time period resulted in a day's worth of development, a week's worth of development, and finally a month's worth of development. In the case of the Apache library which was used for a case study in [4], the same time periods were retained in order to maintain consistency.

For each case study, we carried out mutation testing runs[2] as follows: First using the naïve approach from [4], and then using the control flow analysis technique whilst increasing the depth analysis level until no further methods are identified for mutation. Each mutation testing run was executed three times to ensure consistency in the data, with less than 10% variation observed. Due to the fact that the non-disclosure agreement with our industry partner required that source code does not leave their premises, experiments on the two case studies where done on different machines as shown in Ta-

[2]We also kept the same set of seventeen basic mutation operators from [4], relying on the mutation coupling effect hypothesis stating implying that complex mutants do not give any significant advantage on simple ones.

Table 1: Hardware setup details for each case study

	Lab Machine	**Industry Machine**
CPU	Intel Core i7 2.2 GHz	Intel Core i7 2.93 GHz
RAM	6 GB	8 GB
OS	Windows 8.1 (64 bit)	Windows 7 SP1 (64 bit)
Java	1.7 (64 bit)	1.7 (64 bit)

Table 2: Absolute results obtained for both case studies

Apache Commons CLI					
Experiment	Depth	Mutants	Killed	Score	Time (s)
Day	Trad.	253	110	43%	20
	0	95	45	47%	5
	1	122	58	48%	18
Week	Trad.	340	113	33%	20
	0	183	57	31%	9
	1	210	68	32%	20
Month	Trad.	349	131	38%	16
	0	158	73	46%	6
	1	306	108	35%	19
	2	312	111	36%	20
	3	315	112	36%	21
Industry					
Experiment	Depth	Mutants	Killed	Score	Time (s)
Day	Trad.	954	577	60%	63
	0	57	42	74%	4
	1	182	140	77%	23
	2	191	149	78%	24
Week	Trad.	954	577	60%	63
	0	347	210	61%	25
	1	415	237	57%	39
	2	418	240	57%	39
Month	Trad.	954	577	60%	63
	0	700	374	53%	35
	1	728	399	55%	54
	2	731	402	55%	54

ble 1. The full results of the experiments, including the mutation score, are shown in Table 2.

The following subsections focus on the benefit and the cost of applying static analysis in the context of our case studies, followed by a discussion in the final subsection.

4.1 Benefit: Mutant Increase Analysis

The main concern of adopting the naïve approach in distinguishing the changed part from the unchanged part of a system is that potentially valuable mutants might never be generated — meaning that test suite deficiencies might not be detected simply due to such missing mutants.

Thus, measuring the benefit of introducing static analysis mainly consists of counting the number of mutants which would otherwise not have been tested.[3] Table 3 show the percentage increase in the number of mutants from one depth to another.

[3]Note that we assume that it is useful to generate mutants from methods (possibly transitively) calling changed methods. Thus, we ignore whether or not such mutants are actually killed or not. Such information would only have a baring on the quality of the test suite, not the effectiveness of our technique.

Table 3: Mutant percentage increase according to depth change

Apache Commons CLI			
Experiment	Depth 0→1	Depth 1→2	Depth 2→3
Day	28%	0%	0%
Week	15%	0%	0%
Month	94%	2%	1%
Industry			
Experiment	Depth 0→1	Depth 1→2	Depth 2→3
Day	219%	5%	0%
Week	20%	1%	0%
Month	4%	<1%	0%

Analysing the results horizontally suggests that most changes do not occur deeper than one level below the surface of the call graph, showing significant increase in mutants only when going from depth zero to depth one. Although our case studies show that the influence of a change never reaches beyond a depth of three method calls, this is highly dependent on the topology of the individual system's call graph and the location of the change itself.

Whilst the rows of the tables give us insight into the topology of the call graph — namely its depth and how this is distributed, we hypothesise that the columns of the tables can shed light on the kind of changes in the code churn. More specifically, on how the changes were distributed along the call graph: if the changes are focused along a particular branch of the call graph, then one would not expect to find many affected methods which have not been directly modified. On the contrary, if several unrelated branches have minor changes away from the root, then one would expect to find numerous methods which would have been affected. Naturally, such a distribution depends on numerous factors including the maturity of the project (mature projects would tend to have more minor and unrelated modifications in deeper methods) and on the way the work on the project is managed (an industry team of developers would tend to work more on a feature by feature approach whilst in the case of an open source project, one would expect many unrelated parts of the project to be touched). The observations seem to be corroborated by the disparaging results of the tables where in the case of our industry case study, the changes were more focused on a day by day basis and yet covered most of the system over a month since the system was still being developed. On the other hand, in the case of the open source library the wider breadth of the changes over a month meant that the number of mutants added through static analysis was at its highest point.

4.2 Cost: Execution Time

Compared to the traditional mutation testing approach which attempts to kill all the mutants (irrespective of whether they were affected by the changes), the proposed approach has the advantage of typically excluding a substantial number of mutants from analysis but still has the disadvantage of carrying out analysis on what we consider to be valuable mutants. Effectively, the gains can be characterised as follows:

$$\begin{aligned} \textit{Total Gains} &= \textit{Time(all muts.)} - \textit{Time(selected muts.)} \\ \textit{Actual Gains} &= \textit{Total Gains} - \textit{Time(static analysis)} \end{aligned}$$

More concretely, this means that unless the number of included mutants multiplied by the time required to execute the test suite is high

Table 4: Performance gain from traditional mutation testing

Apache Commons CLI				
Experiment	Depth 1		Depth 2	
Measure	%muts.	speedup	%muts.	speedup
Day	48%	1.1x	48%	1.1x
Week	83%	1.0x	83%	1.0x
Month	88%	0.8x	89%	0.8x
Industry				
Experiment	Depth 1		Depth 2	
Measure	%muts.	speedup	%muts.	speedup
Day	19%	2.7x	20%	2.6x
Week	44%	1.6x	44%	1.6x
Month	76%	1.2x	77%	1.2x

Table 5: Execution time compared to naïve approach

Apache Commons CLI			
Experiment	Naïve	Proposed	
Measure	(depth 0)	depth 1	depth 2
Day	5	18	18
Week	9	20	20
Month	6	19	20
Industry			
Experiment	Naïve	Proposed	
Measure	(depth 0)	depth 1	depth 2
Day	4	23	24
Week	25	39	39
Month	35	54	54

enough to at least counteract the time spent in static analysis, the approach is not beneficial. In fact, referring to the results shown in Table 4, we note that in the case of the *month* time period for the Apache library, where most of the mutants were selected anyway, incremental mutation testing approach was actually slower than traditional mutation testing.

The proposed technique, whilst still generally faring better than traditional mutation testing, when compared to the naïve approach of mutating only directly changed methods, is naturally slower due to two main reasons: the generation of the call graph and the generation and analysis of the extra mutants. Arguably, the time spent on the latter is justifiable by the increased effectiveness, otherwise one should stick to the naïve approach in the first place. However, the time spent generating the call graph is significant when compared to the time required to analyse the call graph: eight seconds for the Apache library case study and twelve for the industrial one. Table 5 provide the total execution time for the naïve and the proposed approach including the time to generate the call graph. Whilst at face value the difference is staggering, when one considers the increase in mutants being considered and the cost of generating the call graph, the numbers add up: For example the seemingly huge nineteen second gap from four to 23 seconds in the industry day scenario is composed of twelve seconds to generate the call graph, and seven seconds which is approximately 219% of four seconds required in the naïve approach.

Finally, considering the very small differences in readings between the values in *depth 1* and *depth 2* columns, notwithstanding the minimal increase in the number of changed methods identified, we note that compared to the cost of generating the call graph, the cost of analysing it to consider deeper methods is negligible.

We conclude this section by noting a number of limitations our analysis currently has:

Runtime resolution of dependencies There are cases where a control dependency is resolved at runtime: one such case is when a method call is resolved polymorphically, while another example would be J2EE XML configurations. At the moment we do not take these into consideration, possibly missing out on a number of useful mutations.

Depth estimation Due to the limited number of case studies, we are not able to estimate the depth required in general to identify all methods requiring mutation. From our experience, in both cases most methods were identified at depth one, very few at depth two, and even fewer at depth three. Whilst it is desirable to have a way of calculating the depth required upfront, we note that the analysis of the call graph for an extra level is negligible compared to the time needed to carry out mutation operators and rerun the test suite for each mutant.

Effectiveness approach with respect to changes demographic While we have made a number of remarks regarding how the distribution of code modification across the call graph may affect the effectiveness of our approach, we believe that this can be explored more in the future. Ideally, one could have an indication of how useful the technique would be before actually wasting time generating the call graph.

Incrementally building the call graph Our current implementation rebuilds the call graph each time the incremental mutation process is carried out. This is a substantial limitation given how expensive it is to build the call graph. In the future, this can be built incrementally by simply updating the call graph to reflect the changes in the code.

5. RELATED WORK

Several related works dedicated to making mutation testing more feasible already exist, each of which falls into one of three categories: "do smarter", "do faster" and "do fewer" [8]. Perhaps incremental mutation testing could loosely fit under "do fewer", although our aim is to split the cost of mutation testing over iterations rather than reducing the number of mutants per se. In this particular paper, our focus was on improving an existing technique rather than develop a new one. To the best of our knowledge, there is no other work on improving incremental mutation testing, so we will focus this section on work from the field of *regression test selection*, our main influence of this work.

In the field of regression test selection, given a program P, the modified version of P, P', and a test suite T, the test case selection problem involves finding a subset of T, T', with which to test P' [16]. This problem has strong parallels with our problem of finding a subset of methods in an evolution of the program, which are likely to produce valuable mutants during mutation analysis.

The literature contains a wide variety of approaches to addressing the problem including amongst others *dynamic program slicing* [1], *data flow analysis* [7], symbolic execution [15] and graph walking [13]. Of these, the latter is most closely related to the work presented here. Graph walking techniques involve the gen-

eration of graph-based internal representations of a program which are subsequently analysed with respect to a certain question. In general, graph walking techniques involve the generation of one of the graph-based representations for both P and P', and then *walking* through the graphs looking for paths which lead to modified code. Once these paths have been identified, test cases which execute control-dependent predecessors of the modified code are selected. In this work, we adopt the same technique albeit to select methods for mutation generation rather than tests for regression testing. Control dependent predecessors of a node N in a control-oriented graph representation of a program are nodes which at some point in the execution have an influence as to whether or not N is executed.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we presented a technique which utilises control flow analysis in order to more precisely characterise the differences between two versions of a program. We also discussed results from two case studies which indicate that the technique does in fact improve the effectiveness of incremental mutation testing, albeit at a cost in performance. Having said that, in most cases, the loss in performance still resulted in a substantial speedup over traditional mutation testing. We also observed that whilst the nature of change and the topology of the call graph will influence performance, in general, the more frequently incremental mutation testing is executed, the more cost-effective it becomes. That is to say that carrying out mutation analysis at the end of each day is more cost effective than carrying it out once a month. This is because a large number of changes would have occurred in the longer time period, resulting in more mutants needing to be generated and analysed in one run.

We acknowledge that the results presented here suffer from threats to validity in the same way that all case study based evaluations do. Two case studies are by no means representative of the entire population of software systems and further studies are required in order to generalise the results. However, we argue that the results have provided some interesting initial insights into the costs and benefits of the present technique.

6.1 Future Work

There are several avenues of exploration to further improve the effectiveness of incremental mutation testing. Firstly, we would like to carry out wider-reaching case studies on projects in both the industry and the open-source world. This should help us form a better understanding about the contexts in which the technique is likely to be beneficial when compared to traditional mutation testing. Secondly, we would also like to consider other types of coupling when analysing the impact of a syntactic change on a system. More specifically, we would like to investigate how considering data flow within the system would affect incremental mutation testing and what (if any) overlap this would have with control flow analysis.

7. REFERENCES

[1] H. Agrawal, J. R. Horgan, E. W. Krauser, and S. London. Incremental regression testing. In *International Conference on Software Maintenance (ICSM)*, volume 93, pages 348–357, 1993.

[2] F. E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, July 1970.

[3] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 402–411, New York, NY, USA, 2005. ACM.

[4] M. A. Cachia, M. Micallef, and C. Colombo. Towards incremental mutation testing. *Electronic Notes in Theoretical Computer Science*, 294:2 – 11, 2013. Proceedings of the 2013 Validation Strategies for Software Evolution (VSSE) Workshop.

[5] M. E. Delamaro, J. Maldonado, A. Pasquini, and A. P. Mathur. Interface mutation test adequacy criterion: An empirical evaluation. *Empirical Softw. Engg.*, 6(2):111–142, June 2001.

[6] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call graph construction in object-oriented languages. *SIGPLAN Not.*, 32(10):108–124, Oct. 1997.

[7] M. J. Harrold and M. L. Soffa. Interprocedual data flow testing. In *ACM SIGSOFT Software Engineering Notes*, volume 14, pages 158–167. ACM, 1989.

[8] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *Software Engineering, IEEE Transactions on*, 37(5):649–678, 2011.

[9] H. M. Jia Y. An analysis and survey of the development of mutation testing. *ACM SIGSOFT Software Engineering Notes*, 1993.

[10] T. C. Lethbridge and R. Lagamiere. *Object-oriented software engineering - practical software development using UML and Java.* MacGraw-Hill, 2001.

[11] G. Meszaroz and A. Wesley. Xunit test patterns: Refactoring test code. *Citado na*, page 27, 2007.

[12] N. Nagappan. *A Software Testing and Reliability Early Warning (STREW) Metric Suite*. PhD thesis, 2005.

[13] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(2):173–210, 1997.

[14] O. Shivers. Control flow analysis in scheme. *SIGPLAN Not.*, 23(7):164–174, June 1988.

[15] S. S. Yau and Z. Kishimoto. A method for revalidating modified programs in the maintenance phase. In *Annual International Computers, Software & Applications Conference (COMPSAC)*, volume 87, pages 272–277, 1987.

[16] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.

Author Index

Bajada, Luke 73
Burchell, John 28

Capiluppi, Andrea 38
Colombo, Christian 73

Demeyer, Serge 1, 9

Granli, William 28

Hammouda, Imed 28
Hassan, Ahmed E. 46
Hayase, Yasuhiro 66
Hayashi, Shinpei 19
Hyrynsalmi, Sami 56

Ishio, Takashi 66

Kamei, Yasutaka 46
Kanda, Tetsuya 66
Knauss, Eric 28

Laghari, Gulsher 1
Leppänen, Ville 56

Matsuda, Jumpei 19
McIntosh, Shane 46
Micallef, Mark 73
Murgia, Alessandro 1

Pérez, Javier 9

Ruohonen, Jukka 56

Saeki, Motoshi 19
Soetens, Quinten David 9

Ubayashi, Naoyasu 46

Yamashita, Kazuhiro 46
Youssef, Ahmmad 38

Zaidman, Andy 9