Original source of publication: What Can Changes Tell about Software Processes? WETSoM'14 June 3, 2014, Hyderabad, India , DOI: Copyrighthttp://dx.doi.or2014 ACMg/10.1145/2593868.2593869978-1-4503-2854-8/1
The original publication is available at http://dl.acm.org/citation.cfm?id=2593869

What Can Changes Tell about Software Processes?

Barbara Russo
Free University of Bozen-Bolzano
Faculty of Computer Science
P.za domenicani, 3
Bozen, Italy
Barbara.Russo@unibz.it

Maximilian Steff
Free University of Bozen-Bolzano
Faculty of Computer Science
P.za domenicani, 3
Bozen, Italy
maximilian.steff@gmail.com

ABSTRACT

Code changes propagate. Type, frequency, size of changes typically explain and even predict impact of changes in software products. What can changes tell about software processes? In this study, we propose a novel method to render software processes by graphs of linked commits as carriers of change information. Mining histories in such commit graphs allows to exploit techniques of graph analysis and coloring that can be used to understand activities in software processes. As application of our method, we analysed colored commit graphs to investigate the presence of large architectural changes and their likelihood of occurrence in bug fixing. For this, we introduced a new measure of architectural change based on hashing and a linear-time kernel for bit-labels graphs. We applied our approach to analyse the evolution of change of Eclipse JDT and Spring Framework.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Measurement techniques; H.2.8 [**Database Applications**]: Data mining

General Terms

Software process, software evolution

Keywords

Software changes, commits, directed graph, architectural change, churn

1. INTRODUCTION

When developers commit their code they must know the effects of their changes. Changes may introduce new changes that can eventually affect the quality of software products. Causality, localization, classification of changes have been matter of intensive research in software evolution and quality. Several technologies that help developers to understand the impact of their changes on code have been proposed. Such technologies are valuable, but suffer from the information used for changes, typically based on churn, i.e., lines added and/or deleted from file. Only recently, other types of data like code dependencies are being investigated, [18].

Changes have effects on software development activities too. Lehman [15], clearly warned about the risk of leaving changes and their complexity uncontrolled: software development processes must adapt to changes as well. Of course, the opposite is also true: development methods and strategies drive code changes. For example, we can sometimes observe an increase of the change rate before a new release is announced.

Mining data to evaluate the effect of changes on software development activities is not so simple. One of the major issues is subjectivity and discretion of developers in reporting software process data. Herzig and Zeller found that 20% of commits were incorrectly assigned to bug fixing as developers delivered changes to accomplish multiple tasks in a single commit, [11]. On the other hand, changes related to one single task may not occur in close temporal proximity. Thus, analysing product or process behavior with measures based on a temporal order of changes (e.g., consecutive changes as in [17]) might not always be the most satisfactory approach. In this work, we try to answer the second issue by modeling software processes with topological graphs of commits. Specifically, the main contributions of this paper are:

- A labeled commit graph. It is a directed graph whose nodes are commits and edges are determined by files changed in commits. The graph can be colored by assigning semantics of different nature to commits (e.g., bug fix commit).
- A method to color commits by large values of change measures. In particular, we introduce a novel measure of architectural change. It uses a linear-time kernel technique to compute distance between architectures of a system over time.
- An extension to graphs and histories of the well-known gamma score [19]. The extension allows to compare histories in graphs
- A proof of concept that our approach can be applied to object oriented projects, as shown by a case study of Spring Framework and Eclipse JDT.

2. RELATED WORK

Class, file, or method changes have been long investigated to determine the past and future state of a software project.

Examples include impact analysis, change classification, and change measurement.

2.1 Impact Analysis.

The impact of code changes is typically studied for causality of changes or prediction of other code measures. For example, German et al. introduced the change impact graph to detect and visualize the propagation of function changes and help developers to localize bugs, [5]. Herzig and Zeller, [10], mined method definitions and calls over time to investigate change causality. Nagappan et al. defined measures of consecutive changes (bursts) to predict defect counts [17]. Tomaszewski et al [21] use the history of previous releases to predict LOC changes of the current release by the number of added and changed methods per class. Lee et al [14] analysed the impact of various changes on the overall system. They identified a number of fine-grained changes and how the impact of these changes could be evaluated before implementation. Both Tomaszewski et al [21] and Lee et al [14] focus on the addition and modification of methods as they aim at predicting code changes at early stages of development. Recently, Wu et al [23] examined eleven open-source systems and their histories of changes. They found power-law distributions for change sizes and structural changes, and that these changes were occurring across the system. Unlike ours, the measure of structural change they introduced refers to local modification to classes' dependencies. As the authors also say, their definition of structural change is affected by how one interprets the local changes and how frequently structural snapshots are captured. Interestingly, the authors discovered long-range correlations in time series of change" from which they inferred a self-organised criticality in the evolution of the systems. They concluded that major changes keep occurring and that the development process should be adapted accordingly. However, their results regarding long-term correlations were then challenged by Herraiz et. al. [9] albeit on a different dataset.

2.2 Change Classification.

Literature mainly focuses on syntactical types or types determined by the task they accomplish. Gall et al [4] introduce logical coupling as change pattern similarity of files over releases. Logical coupling has also been recently considered in Canfora et al [1], where the authors use the Granger causality test on time series to determine whether a change to a file really causes subsequent changes to other files. In a similar vein, Herzig and Zeller [10] have mined method definitions and calls over time to the so-called change genealogies in which previous changes enable and cause later modifications.

Giger et al., [6] classified syntactical changes to predict them in future releases. Hindle co-authored a series of works to classify maintenance activities with the information contained in commit messages. Recently, Hindle et al. further proposed a method to extract maintenance activity types from commit messages, [13]. A recent paper of Cataldo et al [2] compares three types of dependencies among files: synthetic, logical, and work/social dependencies (the last being the number of linked developers working on a file). The paper recommends to consider both logical and work / social dependencies in the estimation of fault proneness of files.

2.3 Change Measurement.

Change measures are defined on code change (diff) at different granularity levels (e.g., files, classes, methods, or commit, versions, releases). The typical measure of change is churn. Churn measures the size of a change set in a commit and has been often employed in prediction models, e.g., [7]. Other measures based on syntactical dependencies have been introduced more recently, e.g., Nakamura and Basili, [18].

While literature has mainly focused on the effects of changes on software product, our work proposes to investigate changes also to understand software process. The manner in which process activities are carried out is reflected in the structure of the history of (sets of) files. We believe that the commit graph as analysis tool has promising applications also beyond the study given in this paper, [20]. As we are going to show, commit graphs can be colored to represent product and process in one single shot.

3. METHOD

The key assumption behind this work is that changes made to complete a developer's task might not occur in close temporal proximity and they might be better related by what in common they change. For this reason, we believe that the development process can be rendered with a topological graph that we can color and examine through its substructures. Existing literature can also be re-read with the use of the graph.

3.1 Commit Graphs

A commit C is a tuple (t, F), where t is a time stamp and F is a set of modules, called change set. A Commit Graph (CG) is a directed graph that consists of:

- a set of commits $\{(t, F)\}$ as nodes
- a set of links $\{L\}$ between commits such that L links C_1 to C_2, $((C_1, L, C_2))$, if and only if
    i. $t_1 < t_2$ and $F_1 \cap F_2 \neq \emptyset$ and
    ii. $\forall(t, F)$ with $t_1 < t < t_2$: $F_1 \cap F_2 \cap F_3 = \emptyset$

Two commits that changed one or more common modules are linked if no other commit has changed any of the modules in between. Fig. 1 shows an example of CG. The total number of commits directly linked to one commit C does not exceed the cardinality of its change set, F.

Modules can have different granularity depending on the type of research we want to perform. We can consider methods, classes, files (if we do not want to distinguish inner and nested classes), or packages. Links can be enriched with further information too. For example, a link can be weighted by the number of modules shared between its commits, or tagged by the developers that changed them. With our definition, nodes can also have different granularity ranging form commits to major versions.

3.2 Commit Histories.

CG is made of a set of commit histories. Studying them can shed some new light on software systems and their evolution. For example, one can study the evolution of logical coupling [4], by studying paths in the full history (FH) of a commit $C = (t, F)$:

$$FH = \{(t_i, F_i) : t_i < t \, F_i \cap F = \emptyset\}$$

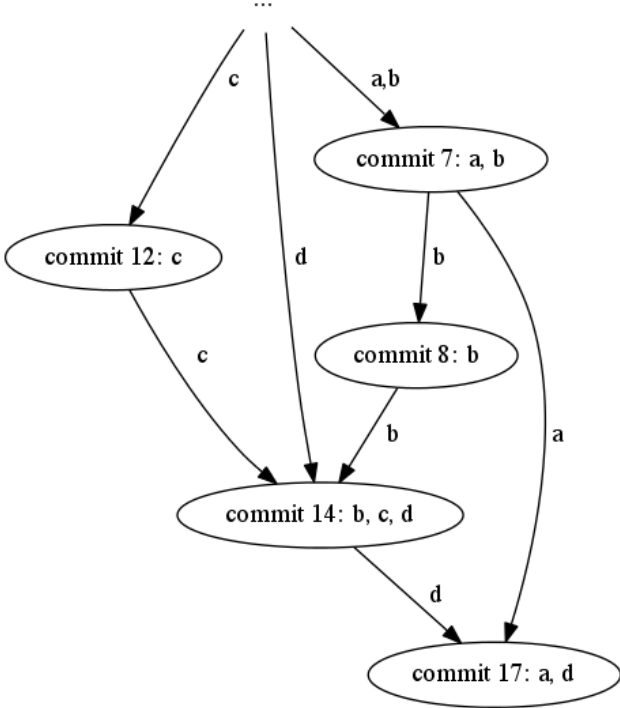

Figure 1: An example of CG.

In this study, we introduce two other types of history: progeny and ancestry. Progeny (Ancestry) is the set of commits that follow (precede) a commit in the commit graph. In this study, we limit progenies and ancestries to two-steps histories of a commit C:

$$PH = \{A|\exists L : (C, L, A) \vee \exists B, L', L : (C, L, B) \wedge (B, L', A)\}$$

Likewise, ancestry is the set of commits of parents and granparents:

$$AH = \{A|\exists L : (A, L, C) \vee \exists B, L', L : (A, L', B) \wedge (B, L, C)\}$$

PH and AH are different from FH as they can include commits that changed files not in change set of C.

Coloring CGs. Nodes in graphs can be given colors to render the development process at different angles. Colors are labels with some given semantics. In this work, we color commits in two ways: by maintenance tasks (using developers' tags) and by their technical nature (using large values of measures of change). In future work, we will investigate whether large values of measure of changes can identify specific development tasks.

3.3 Progeny Score.

The Gamma score is a non-parametric statistical techniques that was introduced by Pelz in 1985 [19] to compare precedence of categorical items (e.g., modification requests [22]) appearing in time series.

We adapt the score to describe the order of occurrence of different types of commits in colored graphs. As a matter of example, in this work, we analyse the probability for a commit of a given color to have at least a commit of a different color in its progeny. For this, we propose the score I :

$$I_{A,B} = \frac{P - Q}{P + Q}$$

where P (Q) is the number of color-A (B) commits with at least one color-B (A) commit in their progeny. With this score, we can examine, for example, whether large architectural changes occur behind large churn changes or whether any type of large change follows up bug fixes.

3.4 Architecture Graphs

Architectural change measures dissimilarity between architectures of a software system at different time instants. In this paper, we represent an architecture as graph in which nodes are architectural components and links are relations among components. As for CGs, components in Architectural Graphs (AGs) can have different granularity and relations different definitions. In this work, we assume that:

- Each class and interface in the source code is assigned a node. Node labels are class names.
- Inheritance, association and use dependencies are links
- External libraries are not included

Architectural Change. Using graph kernel theory, a distance between two AGs, A and A', can be defined as

$$d(A, A') = \sqrt{K(A, A) - 2K(A, A') + K(A', A')}$$

where $K(\cdot, \cdot)$ is a kernel function. When d is 0, no change happens between A and A'; when d is 1 the change is maximal. In other words, the distance d measure the similarity between two architectures.

The most well-known graph kernel is the the random walk kernel also used in Nakamura and Basili, [18]. In this study, we instead propose a linear-time kernel introduced by Hido and Kashima [12] for bit-labels graphs. For this, we first need to map an AG into a bit-labels graph (BG) by hashing. Hashing codes node labels as bit labels. Bit labels are binary arrays of bits of fixed length, $B = \{b_1, ...b_k\}$, where b_i are bits. As node labels in AGs are finite set of discrete values, without loss of generality, we can convert a node label v into a randomly chosen bit label of a given length using a one-to-one mapping function l. Since node labels are distinct in our settings, accidental hash collisions only occurs with probability 2^{-k}. Experimenting with several different lengths k, we found in our graphs $k = 2 * (n + 1)$ where n is minimum bit number such that $2^n \geq card(AG)$.

The Neighborhood Hash (NH) algorithm, [12], incorporates in the hash value information about the adjacent nodes through two operations on the bit labels (XOR and rotation):

$$XOR(B, B') = \{XOR(b_1, b'_1), ..., XOR(b_k, b'_k)\}$$
$$ROT(B, o) = \{b_o, b_{o+1}, ..., b_k, b_1, b_2, ..., b_{o-1}\}$$

If v is a node label and $\{v_1, ..., v_s\}$ are its adjacent nodes, the NH value is obtained by executing the following algorithm:

```
B = l(v)
B_i = l(v_i)
S= {B_1, ..., B_m}
TEMP ← 0
for B_j in S
    TEMP ← XOR(TEMP, B_j)
NH(v) ← XOR(ROT(B),TEMP)
```

Two nodes, v and w, with the same hash value $l(v) = l(w)$, have the same NH value when they have identical neighborhood nodes. Otherwise, they will have different values except for accidental hash collisions. The reason for this is that the hash value is independent of the order of the neighborhood values due to the properties of XOR, [12]. The NH operation for a node v is done in O(k*d) where d is the degree of v.

Using the Hido Kashima kernel has two advantages over random walk kernels as used by Nakamura and Basili, [18]. First, random walk kernels are at least of quadratic runtime complexity. Second, the kernel maintains information about nodes in the nodes themselves. When comparing more than two graphs, this allows for tracing structural changes in nodes across several revisions while retaining information on the reasons for the change. This is not possible with random walk kernels.

Finally, after we mapped A and A' into their corresponding BGs, we align the BGs with radix sort and compute the kernel as the mutual Jaccard index on the sorted bit-labeled nodes:

$$K(A, A') = \frac{c}{n_A + n_{A'} - c}$$

where c is the number of bit-labeled nodes that match in the two BGs and n_A and n'_A are the cardinalities of the two AGs.

To measure the evolution from an initial structure to a final observed one, Nakamura and Basili, [18] introduced the measure of relative similarity between architectures:

$$L(A) = \frac{d(A, A_0) - d(A, A_f)}{d(A, A_0) + d(A, A_f)}$$

where A_0 and A_f are the architectures at the initial and the final observation, respectively. L(A) ranges between -1 and 1. Again relative similarity as measure of architectural change can be analysed at different levels of granularity. We can consider commits, versions, or major releases depending on the type of research. In this work, we use commits. Thus, A is the structure of a system at commit C and L(A) is the architectural change of commit C.

4. RESULTS

As a proof of concept, we applied this method to study the occurrence of large architectural / churn changes in recent histories of bug fix commits of Eclipse JDT[1] and Spring Framework[2]. In both projects we collect commits that relate to the development of a future release. Among these commits we highlighted the major versions. Both projects use a similar release strategy with milestones and release candidates, Fig. 2 and 3.

For JDT, we collected data from 23,880 commits of the CVS trunk repository (January 2002 - December 2004, versions 2.0, 2.1 and 3.0).

For Spring, we collected data from 5365 commits of the Subversion trunk repository (July 2008 - December 2011, version 3.0 and 3.1). For this study, we collected classes and interfaces ignoring all non-Java and test classes reducing the total number of useful commits (Table 1).

From the bug tracking systems - JIRA for Spring and BugZilla for Eclipse JDT - we extracted all bug reports (fixed or closed) that affected the above commits. We did this by matching the bug ID in bug reports and additionally commit logs text similarity as described in the following. Typically, there is no direct connection in either the bug-tracker or the VCS linking entries in both. However, it is considered good practice to note a key or an ID from the bug-tracker in the commit message for the versioning control system to indicate whether a commit is related to a ticket in the bug-tracker. There are several techniques described in the literature to identify commits using identifiers from the bug tracker. We devised an additional matching for not-yet-matched entries. Bug-trackers usually provide a short description of a defect. We use the Levenshtein distance on these descriptions and the messages from the commit log to determine their pairwise similarity. We created sets of pairs by selecting entries from either list that were in close temporal proximity. A few samples were typically sufficient to determine a cutoff value for the similarity to decide which commits to add to the list of bug-fixing commits. We choose the cutoff value in a way to minimise the false positives. The drawback of false positives is arguably higher than that of false negatives: we would label bug-fixing commits that other approaches would probably not do.

Finally, we converted JDT CVS to Subversion repository using cvs2svn[3] and processed both Subversion repositories using SVNPlot[4].

Table 1: Projects descriptive analysis

	No. commits	Period	Major vers.
JDT	23,880	01.2002 - 12.2004	2.0, 2.1, 3.0
Spring	5365	07.2008 - 12.2011	3.0 & 3.1

The period we selected for the two projects illustrates two different maturity stages. Eclipse JDT is in its initial releases. By contrast, the two versions of Spring framework have been selected after the first major public release.

To extract dependencies from Java code, we first used Partial Program Analysis for Java, [3] which creates partial builds from Java source code ignoring unfulfilled dependencies, compile the code into byte-code, and then use Apache BCEL[5] to extract dependencies among classes per single build. Finally, we labelled a commit as "arch" if architectural change happens in it, "churn" if it does not, and "bug fix" if the commit has been labeled so by the committer. The first four rows of Table 2 describe the BGs obtained. As nodes correspond to commits, by comparing Table 1 and 2 we see that there is a substantial number of isolated commits. In these commits, files have been changed only once across the versions considered.

4.1 Jumps

To color the commit graph, we compute jumps of our measures. Jumps are commits that contain large changes, which in principle might indicate some peculiar development activity. We first mapped changes over releases to have a first understanding of the occurrence of jumps. To identify them, we use a typical statistical approach that categorises large values by the distance from the values' distribution mean μ. The distance is measured by multiples of the standard

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

[2]http://www.springsource.org/

[3]http://cvs2svn.tigris.org/

[4]http://code.google.com/p/svnplot/

[5]http://commons.apache.org/bcel/

deviation, σ. In this study, jumps are commit whose churn, architectural change, or both exceed $\mu+\sigma$ of their respective distributions. The last four rows of Table 2 describe jumps in the two systems.

Table 2: Descriptive analysis of BGs. "arch" stands for architectural change, "churn" for churn, "arch & churn" for both.

	Spring	**JDT**
#nodes	3,113	17,474
#links	6,139	42,343
#bug-fix nodes	511	6,564
#arch nodes	1,200	2,145
#jumps	186	685
#arch jumps	61	228
#churn jumps	84	371
#churn & arch jumps	41	86

4.2 Progeny precedence analysis

Fig. 2 and Fig. 3 plot churn, architectural change, and number of bug fixing commits cumulatively over commits of Eclipse JDT and Spring framework respectively. In the figures, we can see that architectural changes do not follow bug fixing activities as churn does in Eclipse. By contrast, in Spring framework architectural changes and churn have a similar trend and opposite to one of the number of bug fixing commits. At a first sight, one would imply that in Eclipse architectural changes are not performed for bug fixing. This is implication might not be completely correct. The two plots can only illustrate trends on individual commits. Changes in commits can originate from preceding changes or affect future changes, though. For example, bug fixing can originate new churn, architectural change or new bug to fix. Gamma score on histories can help understand the precedence among different activities in commits as we show in Table 3. Table

Table 3: Precedence of colored commits

Color A	Color B	**Spring**	**JDT**
churn jump	arch jump	0.15	0.09
bug fix	churn jump	0.23	0.78
bug fix	arch jump	0.22	0.71
bug fix	arch & churn jump	0.10	0.84
arch bug fix	churn jump	-0.01	0.14
arch bug fix	arch jump	0.04	0.06
arch bug fix	arch & churn jump	-0.18	0.39
churn bug fix	churn jump	0.13	0.76
churn bug fix	arch jump	0.07	0.68
churn bug fix	arch & churn jump	-0.20	0.81

3 illustrates the results of the Progeny score applied to the graph colored by the two types of jumps and the bug fixing type. The table shows that architectural and churn changes tend not to synchronise in both systems. In particular, in the first row, churn jumps statistically precede architectural change jumps. Nakamura and Basili [18] hypothesised that if this happens, i.e., if architectural changes is far behind churn growth, the change cost will be high. The next three rows indicate that the probability that jumps occur in progenies of bug fixes is statistically higher than they occur in ancestries (especially for Eclipse JDT). This might suggest that bug fixing has a follow-up effect (e.g., refactoring). The two systems are definitely different in fixing bugs that do not require architectural changes (last row). In Eclipse JDT, bug fix commits that do not have architectural changes likelier precedes large changes in both architecture and churn. Again this is the case in which maintenance costs easier increase.

5. CONCLUSIONS

In this paper, we represent software development processes with colored commit graphs and study histories of changes in these graphs. To color commits as architectural change, we introduce a measure of architectural change. The measure extends the similarity distance between two code architectures of Nakamura and Basili [18], improving it by using an efficient hashing algorithm and a simple kernel function on bit-labels graphs defined by Hido and Kashima [12]. We color graphs with either labels coming from the commit message or jumps of architecture or size change.

Then we build the commit graphs for Eclipse JDT and Spring framework. Using a non-parametric score to compute the order of occurrence of commits of different colors, we highlight the differences in the two projects. A correct interpretation of the differences or a causality analysis requires a deeper investigation of the type of systems and their maintenance process that it is out of scope of this work.

There are few facts we need to reflect on, though. First, bug fix commits can accomplish other tasks. Herzig and Zeller [11] found that 16.6 % of all source files are incorrectly associated to bug fixes and proposed an algorithm to detect and filter out wrong associations. As this noise can have a sever impact on any causality analysis, future work will explore the Herzig and Zeller algorithm on commit histories. Second, our hashing works fine until the module label is modified during the change process, in which case the nodes in the AGs before and after the renaming no longer match. To this aim, we detect such refactored classes in the following way. First, we isolated the classes that were removed in the first and were added in the second commit. Second, we compare the two subsets of classes pairwise using the Normalised Compression Distance, [16]. With several different cut-off values narrowing down possible matches, we manually inspected the suggested matches. While certainly not always recommendable due to the effort required for manual inspection, it did solve the problem in our case.

6. FUTURE WORK AND LIMITATIONS

Commit graphs can be colored in different ways depending on the information available in the repositories. We used jumps to identify specific commits, but other ways can be foreseen. Literature typically use commit labels to determine what activity is performed in a a commit, but labels might not be enough, [11]. Developers usually commit file changes for more than one reason. It is an actual open problem to determine what activities are performed in a single commit. Histories in commit graphs can be inspected and classified to characterise them by the activities performed. In this way, not only a commit but its neighborhood can help to describe the development process. Granger causality [8] can be extended to understand the effect of commit activities in histories. This will be a matter of future work.

As in any proof of concept, there are limitations related

to the choices made to exemplify a method on real data. For example, the definition of jump can be different. We use here the standard deviation unit to identify a jump, but depending on the research purpose, a different threshold can be considered and the graph will change accordingly. Also, the way we labeled commit as bug fixing undergoes the usual limitation. Developers might mislabel their commits or the text similarity we used might produce some false negative and positive. The only means to overcome the issue is to interview project stakeholders. This would be matter of future work.

7. REFERENCES

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

[2] M. Cataldo, A. Mockus, J. A. Roberts, and J. D. Herbsleb. Software dependencies,work dependencies, and their impact on failures. In *Software Engineering, IEEE Transactions on*, Vol. 35, No. 6, pages 864–878, 2009.

[3] B. Dagenais and L. Hendren. Enabling static analysis for partial java programs. In *Proceedings of the Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA '08, pages 313–328, 2008.

[4] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Software Maintenance, 1998. Proceedings., International Conference on*, pages 190–198, 1998.

[5] D. M. German, A. E. Hassan, and G. Robles. Change impact graphs: Determining the impact of prior code changes. *Information and Software Technology*, 51(10), pages 1394 – 1408, 2009. Source Code Analysis and Manipulation, SCAM, 2008.

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

[7] E. Giger, M. Pinzger, and H. C. Gall. Comparing fine-grained source code changes and code churn for bug prediction. In *Mining Software Repositories (MSR), 2011 8th IEEE Working Conference on*, MSR'11, pages 83–92, 2011.

[8] C.W.J. Granger Investigating Causal Relations by Econometric Models and Cross-spectral Methods. In *Econometrica*, Vol. 37, No. 3, pages 424–438, 1969

[9] I. Herraiz, J. M. Gonzalez-Barahona, and G. Robles. Determinism and evolution. In *Proceedings of the international working conference on Mining Software Repositories*, MSR '08, pages 1–10, 2008.

[10] K. Herzig and A. Zeller. Mining cause-effect-chains from version histories. In *Proceedings of the International Symposium on Software Reliability Engineering*, ISSRE'11, pages 60 –69, 2011.

[11] K. Herzig and A. Zeller. The impact of tangled code changes. In *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*, MSR'13, pages 121–130, 2013.

[12] S. Hido and H. Kashima. A linear-time graph kernel. In *International Conference on Data Mining*, ICDM'09, pages 179–188, 2009.

[13] A. Hindle, N. Ernst, M. Godfrey, and J. Mylopoulos. Automated topic naming. *Empirical Software Engineering*, 18(6), pages 1125–1155, 2013.

[14] M. Lee, A. J. Offutt, and R. T. Alexander. Algorithmic analysis of the impacts of changes to object-oriented software. In *Proceedings of the Technology of Object-Oriented Languages and Systems*, TOOLS '00, pages 61–70, 2000.

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

[16] M. Li, X. Chen, X. Li, B. Ma, and P. Vitanyi. The similarity metric. *Information Theory, IEEE Transactions on*, 50(12), pages 3250–3264, 2004.

[17] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy. Change bursts as defect predictors. In *Software Reliability Engineering, 2010 IEEE 21st International Symposium on*, ISSRE, pages 309–318, 2010.

[18] T. Nakamura and V. R. Basili. Metrics of software architecture changes based on structural distance. In *Proceedings of the International Software Metrics Symposium*, METRICS'05, pages 8–, 2005.

[19] D. C. Pelz Innovation Complexity and the Sequence of Innovating Stages. In *Knowledge: Creation, Diffusion, Utilization*, Vol. 6, pages 261–291, 1985

[20] M. Steff and B. Russo. Commit graphs. In *Proc. DAPSE*, pages 4–5, 2013.

[21] P. Tomaszewski, H. Grahn, and L. Lundberg. A method for an accurate early prediction of faults in modified classes. In *Proceedings of the International Conference on Software Maintenance*, ICSM'06, pages 487 –496, 2006.

[22] G. Succi, W. Pedrycz, M. Stefanovic, B. Russo. An Investigation on the Occurrence of Service Requests in Commercial Software Applications. In *Empirical Software Engineering*, Vol. 8, No. 2, pages–197-215, 2003

[23] J. Wu, R.C. Holt, and A.E. Hassan. Empirical evidence for soc dynamics in software evolution. In *Proceedings of the International Conference on Software Maintenance*, ICSM'07, pages 244 –254, 2007.

Appendix

Fig. 2 and 3 compare cumulative plots of architectural change (right y-axis) measured by the relative similarity measure, churn (left y-axis), and number of bug fixing commits. The x-axis plots all the commits before the release of the major versions (including them). commits are ordered and plotted according to their time stamp.

Milestones and release candidates ate reported with the original labels (e.g., 2.0m4 is the release milestone 4 of version 2.0 and 3.0rc2 is the release candidate 2 of version 3.0).

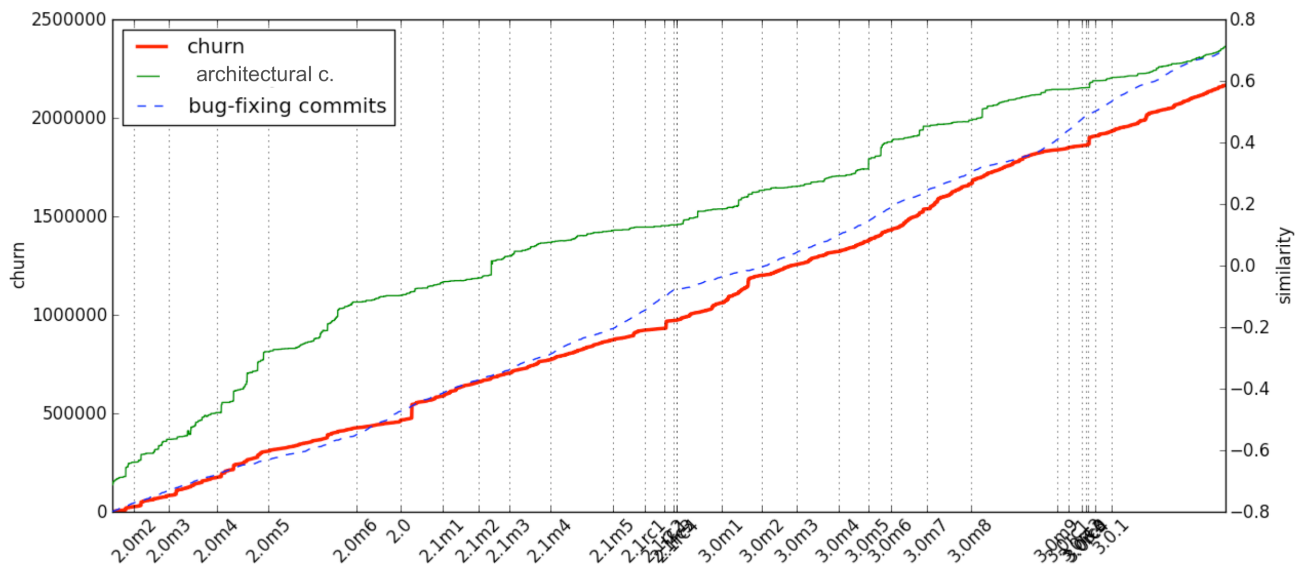

Figure 2: Cumulative architectural change and code churn over commits - Eclipse JDT.

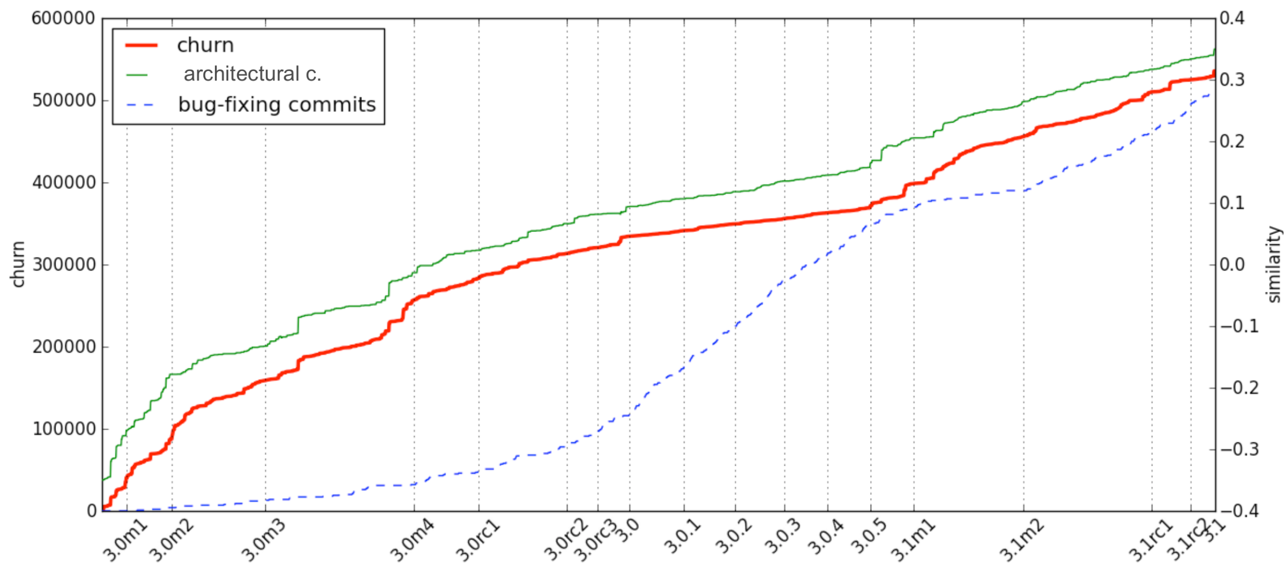

Figure 3: Cumulative architectural change and code churn over commits - Spring.