Effects of Recency and Commits Aggregation on Change Guide Method Based on Change History Analysis

Tatsuya Mori†, Anders Mikael Hagward‡,†, Takashi Kobayashi†

† Graduate School of Information Science & Engineering, Tokyo Institute of Technology,
2–12–1 Ookayama, Meguro, Tokyo, Japan

‡ School of Computer Science and Communication, KTH Royal Institute of Technology,
Stockholm, Sweden

Email: {tmori, anders, tkobaya}@sa.cs.titech.ac.jp

***Abstract*—To prevent overlooked changes, many studies on change guide, which suggest necessary code changes with using co-change rules extracted from a change history, have been performed. These approaches support developers to find codes that they should change but have not been done yet when they decide to commit their changes. The recommendations by existing approaches are adequately accurate when the tools find candidates. However, these tools often fail to detect candidates of overlooked changes. In this study, we focus on two characteristics to increase the opportunity of recommendation to detect more overlooked changes: one is the consideration of recency, i.e., we use only recent commits for extracting co-change rules, and the other is the aggregation of commits for the same task, i.e., we aggregate consecutive commits fixing the same bug. We investigate the effects of our methods on the quality of co-change rules. Experimental results using typical Open Source Software (OSS) show that the consideration of recency can improve the recommendation performance. Our approach can extract more useful co-change rules and recommend more overlooked changes in a higher rank than without the consideration of recency.**

Keywords*–change guide; software repository mining; commit history; software maintenance.

I. Introduction

As the structure of a software program is scaled up, the effort for the ripple effect analysis [1] significantly increases during maintenance, such as bug correction and implementing new features. For the quality of the product, it is important to complete modifications. To prevent overlooked changes, many change guide methods based on static analysis (SA) have been proposed to analyze the scope of change effects [2]. However, these approaches detect many static dependencies include ones unrelated to the change propagation [3]. Further, SA-based approaches fail to find all of necessary dependencies [4]. It cannot find dependencies between codes and non-code elements, such as configuration files and ones through third party libraries.

To overcome the limitations of SA-based change guide methods, studies focusing on the analysis of a software change history in version control system has been performed. These approaches leverage implicit dependencies (aka. logical coupling [5]) extracted from a change history with data mining techniques.

By using association rule mining, an association between code changes are extracted as rules that indicate “if a file is changed, it is highly possible that another file is changed at the same time”. We refer to those rules as *co-change rules*. Zimmermann et al. proposed a co-change recommendation tool, eROSE, that extracts co-change rules from a change history and recommends code elements (e.g., methods or fields) as possible future changes [6]. Their experimental results showed the usefulness of co-change rules for the change guide task. They achieved quite high accuracy for the change recommendation with eROSE. However, the coverage of their recommendation is a few percent; their approach often fails to detect candidates of overlooked changes.

We consider that it is important to expand the coverage for detecting overlooked changes. To increase the opportunity of change guide, we address this low-coverage issue of co-change rules based change guide. In this paper, we propose methods to improve the quality of co-change rules. We focus on two characteristics of change history. One is $rececy$, i.e., how recent the commit was done, and the other is $task$, i.e., which task the commit was related to.

The dependencies that cause change effect become altered along with the project being a long life. That means that the files that had been changed in early term of development might have no dependencies currently. When we use all of the past change histories, we might fail to extract useful dependencies as a consequence of such noise dependencies. We form a hypothesis that we can extract co-change rules strongly related to current changes by considering recency.

When partial changes for a bug fix had been overlooked in past, a developer may have already found these overlooked changes and corrected them. We should treat these change history as a single commit for a bug fix to capture the actual co-change relation. This problem can be generalized as the granularity of commits. Not only for unintended separation, the granularity of commits also depends on the nature of developers and projects. For example, while some developers commit all changes for one task, others may commit changed files in separate revisions for the same task. The difference between developers commits behavior might introduce noise for analysis of change history.

In this study, we investigated effects of the consideration of recency and the aggregation of consecutive commits fixing the same bug on the performance of change guide based on analyzing change history. We formalize our study with the following two research questions:

Copyright (c) IARIA, 2015. ISBN: 978-1-61208-438-1

RQ1 Can we improve the effectiveness of change recommendations with the consideration of recency?

RQ2 Can we improve the effectiveness of change recommendations with the aggregation of consecutive commits fixing the same bug?

The main contributions of this paper are:

- We empirically confirm the usefulness of co-change rules to recommend overlooked changes by using three large OSS projects.
- We indicate that we can recommend more overlooked changes significantly by considering recency as a result of an experiment. Moreover, we can recommend correct overlooked changes in a higher rank than without the consideration of recency.
- We also show that we can improve the performance of recommendation by aggregating consecutive commits fixing the same bug depending on the projects.

Structure of the Paper. Section II discusses the related work. Section III describes the experimental setup. Section IV presents the results of experiments. Section V mentions threats to validity. Section VI closes with conclusion and consequences.

II. RELATED WORK

In this section, we survey the related work in the fields of logical coupling and change guide based on change history analysis. We also survey the related work using some methods that we focus on in our study.

A. Logical Coupling

Gall et al. extracted dependencies between files that have a chance to be changed at the same time by analyzing a change history of a software system stored in version control system, e.g., Concurrent Versions System (CVS) or Git [5]. They called those dependencies “logical coupling.” Logical coupling can represent implicit dependency that can not be extracted by static analysis. Lanza et al. proposed Evolution Radar [7]. This tool integrates both file-level and module-level logical coupling information and visualizes those logical couplings. Alali et al. investigated the impact of temporal and spatial locality on the results of computing logical couplings [8]. Wetzlmaier et al. reported insights about logical couplings extracting from the change history of commercial software system [9]. They indicated resulting limitations and recommend further processing and filtering steps to prepare the dependency data for subsequent analysis and measurement activities.

Zimmermann et al. proposed eROSE [6]. This tool extracts method-level logical couplings and recommends code elements as possible future changes. eROSE extracts logical couplings as association rules by association rule mining. Zimmermann et al. performed an experiment to evaluate a performance of recommendations by eROSE in the scenario that “when a developer decides to commit changes to the version control system, can eROSE recommend related changes that have not been done yet?” As a result of the experiment, $Precision$ of recommendations by eROSE was 0.69. That means that 69% of recommendations were correct. The recommendations by eROSE were adequately accurate. However, $Recall$ was 0.023. (Note that they showed $Recall$ was 0.75 in their paper. However, they calculated this value without the ratio of occurrence of their recommendations. We recalculated actual $Recall$ as the product of their $Recall$ and their $Feedback$.) That means that only 2.3% of overlooked changes could be recommended. The performance of recommendations by eROSE is satisfactory, but we are motivated to recommend more overlooked changes.

B. Change guide based on change history analysis

Kagdi et al. proposed sqminer [10]. This tool uncovers the sequences of changed files spuriously and decreases false recommendations. Gerardo et al. showed that Granger causality test can provide logical couplings that are complementary to those extracted by association rules. They built hybrid recommender combining recommendations from association rules and Granger causality. Their experimental results indicated that the recommender can achieve a higher recall than the two single techniques [4].

C. Consideration of Recency

In the field of data mining related to segmentation in direct marketing, Recency, Frequency, and Monetary (RFM) analysis is often performed [11][12]. RFM means how recently a customer has purchased (recency), how often they purchase (frequency), and how much the customer spends (monetary). On the other hand, an association rule mining is often used in the field of change guide for developers, and this method takes only frequency (how often the files are co-changed in the same commit) into account. The dependencies that cause change effect become altered along with the project being a long life, so co-change rules extracted from very old commits might be useless currently. We form a hypothesis that if we also take recency into account for an association rule mining, we can extract co-change rules strongly related to current changes.

D. Aggregation of Commits related to the same task

McIntosh et al. investigated the dependency between source code files and build files. In their research, they aggregated commits related to the same task for an association rule mining to reduce the noise caused by inconsistent developer commit behavior [13]. They aggregated commits based on information extracted from Issue Tracking System and called those aggregated commits “work item”. They found that work item is a more suitable level of granularity for identifying co-changing software entities rather than a single commit. An aim of their study is detecting the logical couplings between production code changes and build files. On the other hand, an aim of our study is detecting and recommending overlooked changes using logical couplings between source code files, but we think that we can use the same technique for our study.

III. EXPERIMENTAL SETUP

In this section, we describe the tools that we implemented for our experiments, a dataset, experimental settings to address our research questions, and evaluation metrics to evaluate the quality of recommendations.

A. Experimental Environment

We implemented **LCExtractor** for our experiments. LCExtractor extracts co-change rules by using an Apriori algorithm [14]. A co-change rule has a form of “A $\Rightarrow$ B”. The notation “A $\Rightarrow$ B” means “if A is changed, it is highly possible that B is changed at the same time.” “A” and “B” are called the *left-hand*

Copyright (c) IARIA, 2015.

TABLE I. HISTORY OF ANALYZED PROJECTS

Project	# Commits	in Git since
Eclipse JDT	21,378	2001–2014
Firefox	395,466	1998–2014
Tomcat	13,824	2006–2014

side and the *right-hand side*, respectively. The left-hand side is a set of files, and the right-hand side is a file. There are some differences between LCExtractor and eROSE. LCExtractor extracts file-level co-change rules, whereas eROSE extracts method-level change rules. eROSE is an Eclipse plugin. On the other hand, LCExtractor is the tool that spuriously makes the situation, where a file that should be changed is overlooked, and evaluate whether LCExtractor can recommend this overlooked file or not. Therefore, LCExtractor can not recommend to developers actually. However, LCExtractor is superior to eROSE in some ways. LCExtractor can track renamed files and deleted files in a change history. Due to this extension, LCExtractor can recommend renamed files using co-change rules extracted from files before renamed, and exclude deleted files from candidate recommendations. LCExtractor extracts co-change rules by analyzing change history in a modern version control system, Git or Subversion, whereas eROSE extracts co-change rules by analyzing change history in CVS.

Let us explain the process of LCExtractor recommendation. We set the range of target commits, e.g., the latest 1,000 commits. LCExtractor extracts co-change rules using older commits before the target commit. LCExtractor spuriously makes the situation, where one file that should be changed is forgotten to commit, by removing a file from the target commit. Finally, LCExtractor recommends files using co-change rules and evaluate those recommendations. LCExtractor performs above processes iteratively for each target commit in order of old to new. Note that LCExtractor uses commits treated as targets in previous iterations for extracting co-change rules.

This tool was executed on an iMac Retina 5k, Late 2014, with a 4GHz Intel Core i7 and 32GB main memory, running Apple OS X Yosemite.

B. Dataset

For our experiments, we analyzed the change history of three large open-source projects (Table I). We cloned all of the commit histories of those projects as of December 2, 2014, from GitHub.

C. Consideration of recency

To investigate how the consideration of recency affects a performance of change recommendations, we compared the case when we used only recent 5,000 commits older than a target commit for extracting co-change rules, to the case when we used all of the commits older than a target commit. We refer to the latter case as a baseline. Concerning Firefox, we used 20,000 commits older than a target commit instead of all of the commits for the baseline. It is because the total number of commits was very large (about 400,000) and it was difficult for LCExtractor to use all of them for calculating co-change rules.

The Apriori algorithm used in LCExtractor required two parameters: minimum support ($minsup$) and minimum confidence ($minconf$). We set $minsup$ to be 0.0025 for Eclipse and Firefox, and 0.001 for Tomcat. We set $minconf$ to be from 0.1 to 0.9 in steps of 0.1 for each project. As described in Section III-A, we need to set the range of target commits. In this experiments, we use 2,000 commits as target commits for Eclipse, 5,000 commits for Firefox, and 3,000 commits for Tomcat.

D. Aggregation of consecutive commits fixing the same bug

To investigate how the aggregation of consecutive commits fixing the same bug affects a performance of change recommendations, we compared the case when we aggregated consecutive commits fixing the same bug, to the case when we did not aggregate. We refer to the latter case as a baseline. In the former case, we referred to a commit message of each commit and checked if the commit message contains a bug id. If the commit message partially matched one of the following regular expressions, we assumed that the commit was fixing a bug.

- bug[# \t]*[0-9]+
- pr[# \t]*[0-9]+
- Show_bug\.cgi\?id=[0-9]+

If the messages of consecutive commits contain the same bug id, we aggregated them, i.e., we treated them as one commit. In our experiment, we did not take other information of commits (e.g., author or an interval between each commit) when we aggregated them.

In this experiment, we used all of the commits older than a target commit for extracting co-change rules, i.e., we did not consider recency. Concerning Firefox, we used 20,000 commits older than a target commit instead of all of the commits as we did in Section III-C. The settings of two parameters ($minsup$ and $minconf$) and the range of target commits were same as Section III-C.

E. Evaluation Metrics

The most important aim of our study is a prevention of overlooked changes. We used an evaluation setting that was similar to the error prevention setting in [6] to evaluate the quality of recommendations. We used $Precision$ and $Recall$ for the metrics of recommendations. $Precision$ represents the accuracy of the recommendations. $Recall$ represents the ratio that the expected files are recommended. Because our aim is a prevention of overlooked changes, $Recall$ is important rather than $Presicion$. In our experiments, the expected recommendation is only one file. As a result of this experimental setting, it is possible that $Precision$ become low unfairly due to many false positives. To evaluate an accuracy of our recommendations, we also use Mean Reciprocal Rank (MRR). This metric is not used in [6]. The high MRR score means that the expected files are recommended in a higher rank. For example, if most of the expected files place top three recommendations, MRR is higher than 0.33. The definition of the metrics for co-change rules is described as follows.

Let the set of co-change rules be $Rule = \{(l_1, r_1), (l_2, r_2), ..., (l_m, r_m)\}$, where l_i is the left-hand set of files and r_i is the right-hand file described in Section III-A. Let commit history be $Commit = \{com_1, com_2, ..., com_n\}$, where com_i is the set of files. For every changed file $c \in com_i$, let $recom_{i,c}$ be the recommended file set from the changed files

without c, as described below. In this experiments, c represents a overlooked change file.

$$recom_{i,c} = \bigcup_{(l_j,r_j)\in Rule} \begin{cases} r_j & (\text{if } l_j \subseteq (com_i - \{c\})) \\ \emptyset & (\text{else}) \end{cases} \quad (1)$$

For every overlooked change file $c \in com_i$, let $rank_{i,c}$ be the rank of $\{c\}$ in recommendations ranked by $confidence$. The $confidence$ is one of the measure to evaluate the quality of an association rule [15]. If the recommended file set do not contain $\{c\}$, $rank_{i,c}$ is 0. Next, we define $feedback_i$, $precision_i$, $recall_i$, and mrr_i for each com_i (note that $|\{c\}|$ is always 1).

$$feedback_i = \frac{1}{|com_i|} \sum_{c\in com_i} \begin{cases} 1 & (\text{if } recom_{i,c} \neq \emptyset) \\ 0 & (\text{else}) \end{cases} \quad (2)$$

$$precision_i = \frac{1}{feedback_i \cdot |com_i|} \sum_{c\in com_i} \frac{|recom_{i,c} \cap \{c\}|}{|recom_{i,c}|} \quad (3)$$

$$recall_i = \frac{1}{|com_i|} \sum_{c\in com_i} \frac{|recom_{i,c} \cap \{c\}|}{|\{c\}|} \quad (4)$$

$$mrr_i = \frac{1}{feedback_i \cdot |com_i|} \sum_{c\in com_i} \frac{1}{rank_{i,c}} \quad (5)$$

Similar to [6], we calculated $precision_i$ with $feedback_i$ as the denominator, in the sense of "the accuracy of when the recommendation was displayed." If $feedback_i$ was 0 (no recommendation is displayed at this commit), we did not calculate $precision_i$ and excluded this commit from calculating $Precision_M$. Unlike [6], we calculated $recall_i$ without $feedback_i$ as the denominator, in the sense of "the rate of detecting overlooked changes for all commits, regardless of whether of not the recommendation is displayed." Similar to $Precision$, if $feedback_i$ was 0, we did not calculate mrr_i and excluded this commit from calculating MRR. Finally, let $Precision_M$, $Recall_M$ and MRR be the average of $precision_i$, $recall_i$ and mrr_i. Additionally, we define the $F\text{-}measure$ by calculating the harmonic mean of $Precision_M$ and $Recall_M$ to evaluate the performance of recommendation comprehensively because there is a trade-off between $Precision$ and $Recall$.

$$Commit^* = \{com_i | com_i \in Commit, feedback_i \neq 0\} \quad (6)$$

$$Precision_M = \frac{1}{|Commit^*|} \sum_{com_i\in Commit^*} precision_i \quad (7)$$

$$Recall_M = \frac{1}{|Commit|} \sum_{com_i\in Commit} recall_i \quad (8)$$

TABLE II. MAXIMUM $F\text{-}MEASURE$

Project	Considering recency	Baseline
Eclipse JDT	0.137 (minconf: 0.5)	0.063 (minconf: 0.5)
Firefox	0.374 (minconf: 0.8)	0.362 (minconf: 0.8)
Tomcat	0.204 (minconf: 0.4)	0.167 (minconf: 0.4)

$$MRR = \frac{1}{|Commit^*|} \sum_{com_i\in Commit^*} mrr_i \quad (9)$$

$$F\text{-}measure = 2 \cdot \frac{Precision_M \cdot Recall_M}{Precision_M + Recall_M} \quad (10)$$

IV. EXPERIMENTAL RESULT

We performed two experiments. In this section, we describe results of each experiment.

A. RQ1: Can we improve the effectiveness of change recommendations with the consideration of recency?

Figure 1 shows the relations between $Precision_M$ and $Recall_M$, and the relations between MRR and $Recall_M$ for each project with varying $minconf$. The red curve represents the case when we consider recency, and the blue one represents a baseline.

Figure 1.(a), Figure 1.(c), and Figure 1.(e) show that $Recall_M$ increased with the consideration of recency although $Precison_M$ slightly decreased in all projects. As we aim to find more overlooked changes rather than to make the recommendations more accurate, that is a good result. Particularly regarding Eclipse, $Recall_M$ significantly increased. In Figure 1.(a), a maximum $Recall_M$ is 0.28 with the consideration of recency whereas a maximum $Recall_M$ of the baseline is 0.11. That means that we could detect 2.5 times more overlooked changes by considering recency than the baseline.

As a result of considering recency, $Precision_M$ is slightly decreased. It is because the number of recommendations increased with consideration of recency, i.e., many false positives decreased $Precision_M$ even if the set of recommendations contained a expected recommendation. However, in the viewpoint of MRR, the recommendations with consideration of recency were more accurate than the baseline. In Figure 1.(b) and Figure 1.(d), MRR clearly increased with consideration of recency. That means that we could recommend overlooked changes in a higher rank with consideration of recency than the baseline. Regarding Tomcat, shown in Figure 1.(f), we could not improve MRR with consideration of recency. We consider that it is because MRR was already high without consideration of recency.

Table II shows a maximum $F\text{-}measure$ of each project in the case of considering recency and a baseline. For all of the projects, a maximum $F\text{-}measure$ achieved by consideration of recency is higher than the baseline. That means that the quality of the recommendations by consideration of recency was comprehensively better than the baseline.

Copyright (c) IARIA, 2015. ISBN: 978-1-61208-438-1

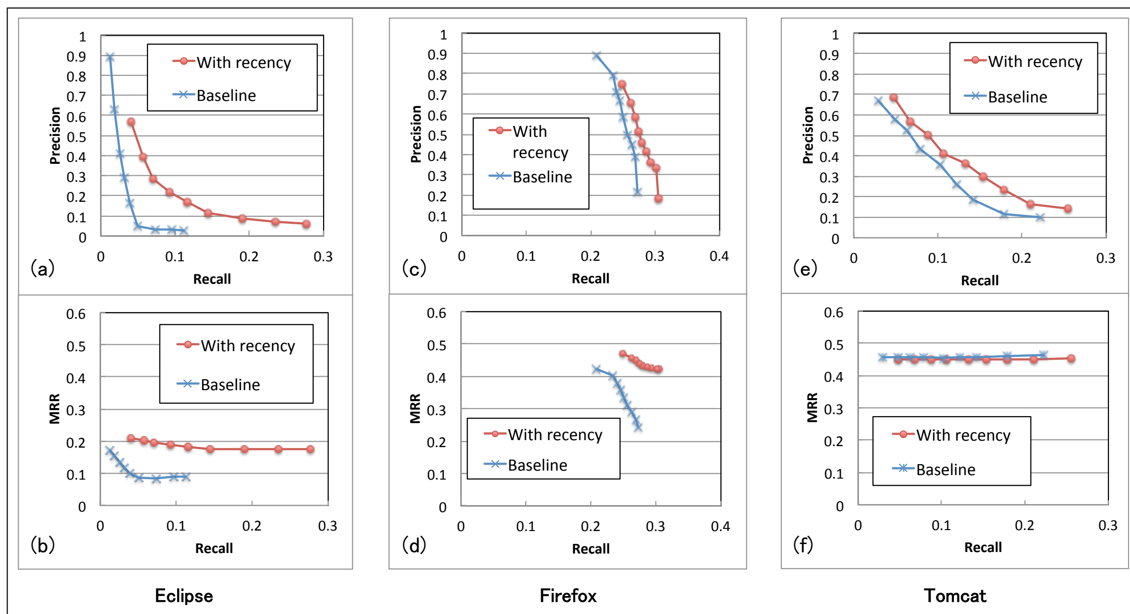

Figure 1. Performance of recommendations by considering of recency and baseline with varying $minconf$.The upper graphs show a relation between $Precision_M$ and $Recall_M$. The lower graphs show a relation between MRR and $Recall_M$.

The answer to RQ1 is **Yes**. If we consider recency, we can extract useful co-change rules that are not able to be extract without consideration of recency. Therefore, $Recall$ of recommendations increase with consideration of recency. Although $Precision$ slightly decrease, we can recommend overlooked changes in a higher rank than without consideration of recency.

B. RQ2: Can we improve the effectiveness of change recommendations by the aggregation of consecutive commits fixing the same bug?

Figure 2 shows the relations between $Precision_M$ and $Recall_M$ for each project with varying $minconf$. The yellow curve represents the case when we aggregate consecutive commits fixing the same bug, and the blue one represents a baseline.

In Figure 2.(a) and Figure 2.(c), we could not improve both $Recall_M$ and $Precision_M$. However, regarding Firefox, shown in Figure 2.(b), $Recall_M$ increased with the aggregation of consecutive commits fixing the same bug. That means that we could extract useful co-change rules that were not able to be extract without aggregation of consecutive commits fixing the same bug depending on projects. As a result of an additional investigation, it is reveal that the number of commits used for extracting co-change rules drastically decreased by aggregating for Firefox (from 20,000 to 15,918), whereas the number of those slightly decreased by aggregating for Eclipse (from 21,378 to 21,098) and Tomcat (from 13,824 to 13,661). We found that the effect that we aggregate consecutive commits fixing the same bug depended on a nature or commit policy of the project.

Table III shows a maximum F-$measure$ of each project in the case of aggregating consecutive commits fixing the same bug and a baseline. Regarding Firefox, a maximum

TABLE III. MAXIMUM F-$MEASURE$

Project	Aggregating commits	Baseline
Eclipse JDT	0.063 (minconf: 0.5)	0.063 (minconf: 0.5)
Firefox	0.414 (minconf: 0.8)	0.362 (minconf: 0.8)
Tomcat	0.167 (minconf: 0.4)	0.167 (minconf: 0.4)

F-$measure$ achieved by aggregation of consecutive commits fixing the same bug is higher than the baseline. Regarding Eclipse and Tomcat, maximum F-$measure$ of two cases are same because we could not improve both $Recall_M$ and $Precision_M$ as previously described. That means that the quality of co-change rules can be improved by aggregation of consecutive commits fixing the same bug in some cases. Moreover, we also found that aggregation of commits did not affect the performance of recommendation in a negative way.

The answer to RQ2 is, in some cases, **Yes**. If we aggregate consecutive commits fixing the same bug, we can extract useful co-change rules that are not able to be extract without aggregation of commits depending on projects. Even if we can not extract more useful co-change rules by aggregation of commits, there is no harmful effect.

V. THREATS TO VALIDITY

Threats to internal validity relate to errors in LCExtractor and parameter settings. We have carefully checked our code, however still there could be errors that we did not notice. In our experiments, we set $minsup$ to be 0.0025 for Eclipse and Firefox, and 0.001 for Tomcat. It is possible that those values were not appropriate. At the moment, we have no method that decide an appropriate $minsup$ prior to performing experiments.

Copyright (c) IARIA, 2015. ISBN: 978-1-61208-438-1

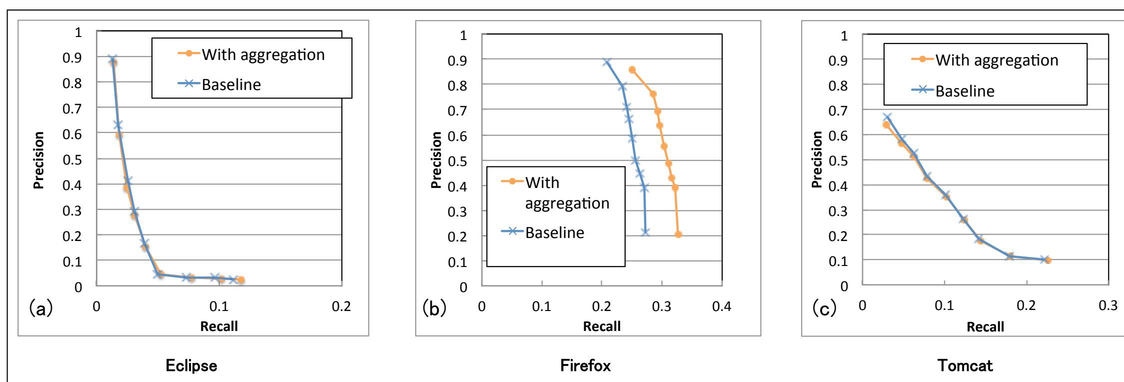

Figure 2. Relation between $Precision_M$ and $Recall_M$ by aggregating of commits and baseline with varying $minconf$.

Threats to external validity relate to the generalizability of our results. We have analyzed 3 different projects. In the future, we plan to reduce this threat further by analyzing more change histories from additional software projects.

Threats to construct validity relate to the experimental settings. We defined using recent 5,000 commits older than a target commit for extracting co-change rules as consideration of recency in the first experiment. However, we did not perform experiments with changing the number of commits for extracting co-change rules. In the future, we plan to reduce this threat further by performing experiments with changing the number of commits used for extracting co-change rules. In the second experiment, we aggregated commits based on only bug id information extracted from commit messages. If we extract more information from Issue Tracking System or Bug Tracking System and use them, we might aggregate commits more appropriately.

VI. Conclusion and Future Work

Numerous studies for supporting developers to find necessary code changes with using co-change rules extracted from the change history have been performed. However, the scope of overlooked changes that existing tools can recommend is limited. In this paper, we focused on the consideration of recency and the aggregation of consecutive commits fixing the same bug. We investigated how they affected the performance of recommendations by using typical OSS (Eclipse, Firefox, and Tomcat). As a result of experiments, we could recommend more overlooked changes by considering recency. We also could recommend correct files in a higher rank than recommendations without consideration of recency. Concerning the case when we aggregated consecutive commits fixing the same bug, we found that the performance of recommendations can be improved depending on projects.

In the future, we plan to perform experiments using more change histories from additional software projects to generalize our theory. In this paper, we indicate that we can improve the performance of recommendations by considering recency. However, we suppose that we can not extract useful co-change rules if we use the small set of commits, e.g., only 10 commits older than a target commit. We plan to investigate how many recent commits are sufficient to extract useful co-change rules.

Acknowledgment

This work is partially supported by the Grant-in-Aid for Scientific Research of MEXT Japan (#24300006, #25730037, #26280021).

References

[1] S. S. Yau, J. S. Collofello, and T. MacGregor, “Ripple effect analysis of software maintenance,” in Proc. COMPSAC’78, pp. 60–65.

[2] L. C. Briand, J. Wust, and H. Lounis, “Using coupling measurement for impact analysis in object-oriented systems,” in Proc. ICSM ’99, pp. 475–482.

[3] M. M. Geipel and F. Schweitzer, “Software change dynamics: evidence from 35 java projects,” in Proc. FSE 2009, pp. 269–272.

[4] 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 Proc. ICSM 2010, pp. 1–10.

[5] H. Gall, K. Hajek, and M. Jazayeri, “Detection of logical coupling based on product release history,” in Proc. ICSM ’98, pp. 190–198.

[6] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, “Mining version histories to guide software changes,” IEEE TSE, vol. 31, no. 6, 2005, pp. 429–445.

[7] M. D’Ambros, M. Lanza, and M. Lungu, “The evolution radar: Visualizing integrated logical coupling information,” in Proc. MSR 2006, pp. 26–32.

[8] A. Alali, B. Bartman, C. D. Newman, and J. I. Maletic, “A preliminary investigation of using age and distance measures in the detection of evolutionary couplings,” in Proc. MSR 2013, pp. 169–172.

[9] T. Wetzlmaier, C. Klammer, and R. Ramler, “Extracting dependencies from software changes: an industry experience report,” in Proc. IWSM-MENSURA 2014, pp. 163–168.

[10] H. Kagdi, S. Yusuf, and J. I. Maletic, “Mining sequences of changed-files from version histories,” in Proc. MSR 2006, pp. 47–53.

[11] P. C. Verhoef, P. N. Spring, J. C. Hoekstra, and P. S. Leeflang, “The commercial use of segmentation and predictive modeling techniques for database marketing in the netherlands,” Decision Support Systems, vol. 34, no. 4, 2003, pp. 471–481.

[12] J. A. McCarty and M. Hastak, “Segmentation approaches in data-mining: A comparison of rfm, chaid, and logistic regression,” Journal of business research, vol. 60, no. 6, 2007, pp. 656–662.

[13] S. McIntosh, B. Adams, T. H. Nguyen, Y. Kamei, and A. E. Hassan, “An empirical study of build maintenance effort,” in Proc. ICSE 2011, pp. 141–150.

[14] P. Bondugula, Implementation and Analysis of Apriori Algorithm for Data Mining. ProQuest, 2006.

[15] R. Agrawal, T. Imieliński, and A. Swami, “Mining association rules between sets of items in large databases,” in ACM SIGMOD Record, vol. 22, no. 2, 1993, pp. 207–216.

Copyright (c) IARIA, 2015. ISBN: 978-1-61208-438-1