Proceedings of the 14th IASTED International Conference
Software Engineering (SE 2016)
February 15 - 16, 2016 Innsbruck, Austria

A CHANGE GUIDE METHOD BASED ON DEVELOPERS' INTERACTION AND PAST RECOMMENDATION

Akihiro Yamamori and Takashi Kobayashi
Dept. of Computer Science, Grad. School of Information Sci. & Eng., Tokyo Institute of Technology
2–12–1, Ookayama, Meguro-Ku, Tokyo, 152–8552 Japan
email: yamamori@sa.cs.titech.ac.jp, tkobaya@cs.titech.ac.jp

ABSTRACT

In this paper, we propose a change guide method based on the past developers' activity that consists of read and write access records of artifacts. In our proposed method, we calculate candidates of next change recommendation considering the history of its recommendations. We define "cumulative likelihood" to enable the method to recommend the appropriate candidates when a change propagates more than one code elements. A case study using interaction history logs from 15 participants showed the improvement of the accuracy of the method-level change recommendation.

KEY WORDS

software maintenance, change guide, interaction history, software repository mining

1 Introduction

Software is continuously changed for various reasons such as adding new features and fixing bugs after release. It has been a serious problem to find a complete set of code fragments that have to be changed while detecting complex dependencies in the program. To improve the quality of the product, it is important to detect all of the source code that have to be changed during maintenance, whose technique is called "change guide" [1][2]. Static analysis based approaches were proposed to detect the scope of change effects [3]. However, these approaches involve the analysis of many dependencies, not all of which cause change propagation [4]. Further, some of these dependencies cannot be found by static analysis [5].

To overcome the limitations of change guide methods that use static analysis, studies focusing on developers' past process information were performed to guide developers during maintenance [6, 1, 7]. These methods estimate the propagation of changes from the developers' past change history on the assumption that the behavior of some developers is similar to that of other developers in the past.

Most of these studies involving developer's past activity used a version control system. It preserves changes between commits. However, it lacks information of the precise order of changes between commits and the entities that were looked by and not changed by the developer. Instead of the version control system, some recent studies used a developer's interaction data including the developer's interaction with integrated development environments (IDE), Web browsers, and other kinds of development tools [8].

To improve the efficiency of the change recommendation process, we previously proposed change guide method using this interaction history including fine-grained information about changes and references to the source code [9]. This study used the reference to an entity between changing entities as 'change context' and introduced the Change Guide Graph (CGG) that is created by learning developers' past interaction histories. It has implicitly made an assumption that the entity that should be changed in the developer's working context can be determined as only one. However, changes of a program often propagate two or more entities at the same time in the actual software development. In this case, this method correctly guides an entity to change, then fails to guide other entities because it will recommend different entities based on the working context that has been changed with the previous change.

In this paper, to overcome the limitation that only one entity of change guide can be used at once, we propose a method that accumulates the likelihoods calculated from the past changes, along with the likelihood calculated from the latest change. We made an assumption that the change candidates displayed recently are also useful as same as the recommendation displayed now, and this method aims to improve the effectiveness of change guide by accumulating likelihoods of past recommendation.

We evaluate our proposed method by the experiment with the interaction date of 15 persons. The experiment shows that accumulating likelihoods is effective in the method-level change recommendation and we should accumulate up to fifth latest likelihoods. It also shows the improvement compared to the previous method by the nDCG (normalized Discounted Cumulated Gain) value which is improved in 34.2% in the method-level recommendation.

The remainder of this paper is organized as follows: Section 2 discusses related works. Section 3 introduces our previous research that proposed a change guide method based on the interaction history. Section 4 points out the limitation of our previous research and propose a change guide method using cumulative likelihood. Section

DOI: 10.2316/P.2016.835-012

5 evaluates our method and discusses the occasions that our proposed method can improve the accuracy of recommendation. Section 6 closes with conclusion and consequences.

2 Related Works

2.1 Change Guide Using Static Analysis of Source Code

The studies using the static analysis of source code were performed to detect the range that changes propagate [3]. By analyzing the source code, we can obtain the dependencies such as method invocations, class inheritances, and field references.

However, there are too many such dependencies in a software and not all of them are involved in change propagation. Geipel et al. [4] were reported that half or more of the dependencies between classes are not involved in the change propagation and a fraction of dependencies causes most of the change propagation. Canfora et al. [5] were also reported that there are some dependencies that the static analysis cannot search. Therefore, the difficulty lies in the change guide using only the source code's dependencies.

2.2 Methods of Mining Revision History

To overcome the limitation of the change guide method using static analysis, other studies were performed that uses the history of the changes to the software artifacts and guides the necessary changes in the maintenance phase.

Gall et al. [6] proposed a method that analyzes the revision history stored by the version control system such as CVS and find the logical coupling, the relation of the files that are likely to be changed together.

Zimmermann et al. implemented a tool, eROSE [1] that can detect such logical coupling in method granularity. When the developer changes some methods, it suggests the methods that have to be changed together. Kagdi et al. proposed another tool sqminer [7] that virtually calculates the time sequence of the changes, which the concept of logical coupling ignores, and improves the accuracy of the change guide. Gerard et al. [5] applied Granger causality test and extracted association rules from revision history of version control system, which can predict a set of change couplings complementary.

These existing studies use the revision history of the version control system, such as CVS, Subversion, and Git. However, there are two limitations. First, the information of changes of artifacts is recorded only when the developer commits, so the information of precise time that change occurred are not recorded. Second, the revision history only stores the changes, so the information of artifacts referenced by the developer is not recorded, which can be the source of the change guide as well as the changed artifacts.

Table 1. List of tools that can output the fine-grained interaction history

tool name	interaction kind
Mylyn [11]	open files, propagate folders, access to files, classes, methods
PLOG [9]	access to files, changed or not, method name at the cursor, content of stdout, stderr.
FLUORITE [12]	insertion and deletion to files, line of code, execution, etc.
CodingTracker [13][14]	text editing, file editing, refactoring, compare editors, interact with VCS, JUnit test runs, start-up, options, etc.
DFLOW [15]	viewing and editing code, software inspection etc.
IDE++ [16]	key strokes with key name, execution, refactoring, save, etc.

2.3 Methods of Mining Interaction History

To make use of the finer-grained developers' activity history compared with the history from the version control system, methods of storing and mining interaction history were proposed [10]. The concept of interaction history is that storing not only the changes of the files, which the version control system stores, but also the developer's interactions including developer's choices and references, and it also stores the time series of the interactions.

There are several tools for recording the interaction history. Table 1 shows a list of tools that capture and output the interaction history.

Kersten et al. made an Eclipse plug-in, "Mylar" (currently renamed "Mylyn") [11] to show summarized artifacts to developers through the IDE. Mylyn can automatically record and display a developer's interaction history. These histories have been recorded in the Mylyn development project since 2007 and are available on *Bugzilla*[1], an issue tracking system. We previously developed PLOG [9] which records the time and time series of developers' code references and editing, and also records runtime exceptions raised at runtime. Yoon et al. developed FLUORITE [12] which records the insertions and deletions to files, the line of code and the number of nodes of the abstract syntax tree. It visualizes transitions of those and developers can use this to see how the software is growing. Negara et al. developed an interaction history recording tool, CodingTracker [13]. It records 38 different kinds of code evolution events including Eclipse's refactoring interaction. They analyzed the interaction history recorded by CodingTracker and discovered frequent code change patterns in [14]. Minelli et al. developed an interaction history recording tool, DFLOW [15], which is specialized in Pharo, an IDE for Smalltalk development. It records

[1] https://bugs.eclipse.org/bugs/

33 kinds of detailed events including inspecting interaction which is peculiar to Smalltalk. They analyzed the interaction history recorded by DFLOW and PLOG to clarify developer's comprehension steps in [15]. Gu et al. developed IDE++ [16], which records 44 kinds of interaction events in very fine granularity, such as key strokes.

Zou et al. defined interaction coupling as the relation between files that are frequently reference-switched with each other [17] to characterize the maintenance tasks. Robbes et al. also proposed similar logical coupling in the fine-grained interaction history [18]. They evaluated various change prediction methods and showed that predictions based on the most recently changed files are the most accurate [19]. Maalej et al. recorded the interaction histories of various development tools and used these to recommend the development tool that the user should use next [20]. Roehm et al. collected code change, Web search, and compile error histories, and produced a representation of the steps taken by a developer when resolving problems by applying the hidden Malkov model [21].

We previously proposed a change guide method using the interaction history that contains changes and references to the artifacts [9]. We used the references between two changes as the context of the changes (See Section 3 for details).

3 Previous Research: Change Guide Graph

In this paper, the **interaction event** is defined as developers' changing or referring activity to software artifacts such as source code elements, documents. In the file-level, an interaction event includes information of the period between opening and closing the file. In the method-level, it includes the period between the cursor entering and leaving the method definition.

We previously proposed a method for the change guide [9], which is performed in three steps

1. Capture interaction history (Section 3.1)
2. Generate the Change Guide Graph (Section 3.2)
3. Predict the next change according to the Change Guide Graph (Section 3.3)

3.1 Capture Interaction History

Using PLOG [9], an Eclipse plug-in, we collect developer's interaction events. The interaction events contain following four kinds of information. In this paper, we call the chronologically-ordered sequence of interaction events **interaction history**.

`name`	The name of the interacted software artifact (eg file or method name)
`start`	The starting time of interaction
`end`	The ending time of interaction
`type`	Whether "Change" or "Reference"

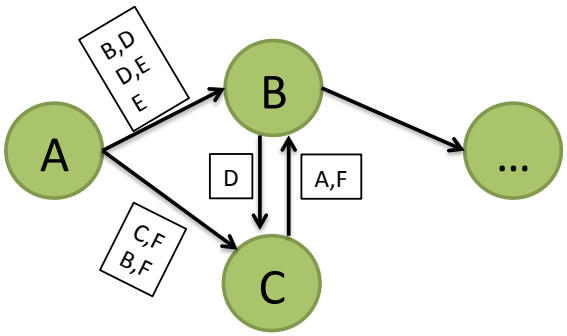

Figure 1. An example of CGG

3.2 Generate the Change Guide Graph

The **Change Guide Graph (CGG)** is a prediction model for change propagation generated by mining interaction histories. The Change Guide Graph is described as a directed graph that its nodes and edges stand for the software artifacts and the context of change propagation, respectively.

The CGG is generated in the following three steps:

1. Cleansing the interaction history
2. Generate "attributed change sequence (ACS)" from the interaction history
3. Generate CGG from the ACS

The detailed procedure of step 1 is described in [9].

In step 2, we generated an **Attributed Change Sequence**. ACS is described as a sequence of changes with attributes that described as a vector of scores of views. For example, if the developer views artifacts A and B before he/she changes an artifact C, ACS will be a change C with an attribute vector of A and B. See [9] for further explanation about generating ACS and scoring views.

In step 3, we generated a Change Guide Graph, a directed graph of which nodes represent changes and edges represent the context of two changes. Figure 1 is an example of CGG. Each of edges has one or more attributes of files with their scores. We generated the graph by iterating following steps:

1. Pick up every consecutive two changes from the ACS.
2. Append the attribute score vector of the earlier change to the edge that directs from the earlier change to the later change. If the edge does not exist, we create an edge and set the attribute score vector of the earlier change.

From these steps, we could store the information of change propagation with the contexts of views from the interaction history.

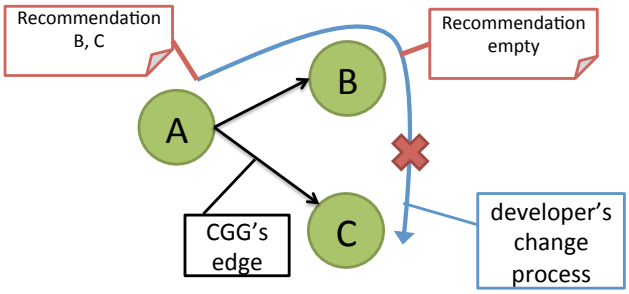

Figure 2. There is no edge $B \rightarrow C$ and this CGG cannot recommend changing C after changing A and B

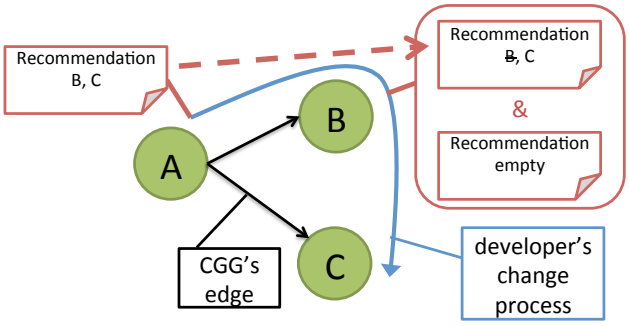

Figure 3. CGG can recommend C after changing A and B because the recommendation after changing A remains after changing B

3.3 Predict the next change according to the Change Guide Graph

When a change event c occurs, we calculated the likelihood $lh_e(c)$ for each edge $e \in E'$ by following.

$$lh_e(c) = \sum_{p_e \in P_e} \{(1-\alpha)+\alpha(conMatch(p_e, p_c))\}$$

Where E' is a set of edges directed from a changed node in the CGG. P_e is the set of contexts of the edge e and p_c is the context of the change. $conMatch(p_e, p_c)$ represents the matching strength of the edge context and the latest change context, which is calculated by a dot product of two vectors p_e, p_c of context information.

The parameter α represents the degree of concerning the context information. If $\alpha = 0$, the $lh_e()$ is independent to $contextMatch()$. The $lh_e()$ is equal to the number of contexts of the edge. If $\alpha = 1$, the $lh_e()$ is fully dependent on $contextMatch()$. In this case, the $lh_e()$ is zero if there is no common context between the edge and the change.

Finally, we guided that the terminal nodes of the edges are the artifacts which should probably be changed. The recommended artifacts were arranged in descending order of the $lh_e()$ of its edge.

3.4 Limitation of the previous method

The previous method assumed that some changes will propagate only one artifact and calculates the likelihood based on the latest one change event. However, changes usually propagate more than one artifact and developers want to change these artifacts.

Assume the source code that change of artifact A may propagate either artifacts B and C, and [A, B] and [A, C] were changed in this order individually in the past maintenance tasks. The edges $A \rightarrow B$ and $A \rightarrow C$ are generated in the CGG in the previous method. The edges $B \rightarrow C$ and $C \rightarrow B$ are not generated based on the past maintenance tasks (Figure 2). The change recommendation after changing artifact A is artifacts B and C. In some cases, developers should change both B and C in this situation. However, the change recommendation will not contain C after the developer changes B because there is no edge $B \rightarrow C$ in the CGG. It is the same if the developer changes C first because there is no edge $C \rightarrow B$ in the CGG.

4 Change Recommendation Based on Cumulative Likelihood

To overcome the limitation of the previous method, we propose a method that can cope with the situation that a change propagates more than one artifact.

The method of the previous method searches only one node of the artifact that has just changed from CGG and explores the edges starting from this node. This causes a limitation mentioned above. Therefore, we use not only the likelihood based on the latest change event but also the change events of second latest and earlier, and we sum up the likelihoods of the edges and make the change recommendation based on the summed likelihood. We call this likelihood "**cumulative likelihood**".

We use the example described in Section 3.4 again. Our extension does not change CGG from the previous method (Figure 3). When the developer changes artifacts A and B in this order, our method will recommend C because it uses not only the likelihood calculated by the change event B (the latest change) but also by the change event A (the second latest change). The cumulation of these likelihoods enables to recommend C as the next change candidate.

Considering cumulating past likelihoods, the likelihood calculated from the latest change event is the most important for the change recommendation. The earlier a change event is, the likelihood calculated from the change event is less important. In this study, we cumulate the likelihoods with a weight.

We define a function for calculating the cumulative likelihood for each artifact (candidate to change) v by following.

$$CL_v(c) = lh_v(c) + \sum_{i=1}^{n} weight(i) * lh_v(c_i)$$

Table 2. Summary of the tasks

Participant	Summary of assigned task
A	renovate Piranha Plant (spits fire)
B	add an item (reverse the arrow key)
C	change Fire Flower to Hummer
D	change Goomba to Spiked Goomba
E	add an item (Raccoon Mario)
F	add an item (10 coin block)
G	add an item (random block)
H	change Fire Flower to Ice Flower
I	change Fire Flower to Bob-omb
J	renovate the behavior of Koopa Troopa
K	add an item (1-Up Mushroom)
L	add an item (Poison Mushroom)
M	add a new action to Mario (Able to punch enemies)
N	renovate the behavior of Koopa Paratroopa
O	add an item (Star)

Where $lh_v(c) = lh_e(c)$ if e stretches from c to v. c_i is the ith latest change. n is the number of changes that are cumulated [2]. $weight(i)$ is the weight for the likelihood of the change event c_i i.e. $lh_v(c_i)$ which is discussed in the case study section. Note that we cumulate the past $lh_v(c)$, not the $CL_v(c)$, so any likelihoods are not cumulated twice even when $n > 2$.

5 Case Study

5.1 Target Software and Interaction History

For collecting the interaction history, we used "PLOG" [9] that is a tool for collecting and storing the interaction history. PLOG was implemented as a plug-in of Eclipse, an Integrated Development Environment for Java. We selected a Java clone implementation of "Super Mario Bros."[3] as a target software for collecting the interaction history, which has 48 classes and 7000 lines of codes. We hired 15 students as case study participants and assigned a different task of adding a new feature for each participant (Table 2).

We forced participants to use "PLOG" and we obtained 15 sessions of interaction histories in both file-level and method-level. Table 3 shows the length (the number of changes) of the attributed change sequence that is generated from obtained interaction history (See Section 3.2).

5.2 Evaluation Metrics

We selected nDCG (normalized Discounted Cumulative Gain) [22] as an evaluation metric for the accuracy of our method. Robbes et al. [19] used it for evaluating their change guide method. nDCG is originally designed to measure the performance of a recommendation system based on the graded relevance of the recommended entities. It varies from 0.0 to 1.0, with 1.0 representing the ideal ranking of the entities.

In this case study, we calculated nDCG iteratively for each change recommendation and treated the arithmetic average of the nDCG as the evaluation metrics for this case study. We also used Wilcoxon signed-rank test to confirm that the variation of nDCG between using the cumulative likelihood and using the likelihood with no cumulation was statistically significant.

[2]The change events are represented as $\{c_n, c_{n-1}, \cdots, c_1, c\}$ when arranging events in chronological order

[3]https://mojang.com/notch/mario/release.zip

Table 3. Summary of the length (the number of changes) of obtained attributed change sequence (ACS)

Participant	Length of the change sequence	
	file-level	method-level
A	23	37
B	19	34
C	7	20
D	60	75
E	32	91
F	19	34
G	3	33
H	13	21
I	40	81
J	7	12
K	9	11
L	11	19
M	150	306
N	36	71
O	13	49

5.3 Parameters

We investigated the impact of the parameters, the weight of the ith latest likelihood $weight(i)$ and the number of cumulating change events n. We varied n from 1 to 9. We selected three fixed values and two functions decreasing as old for $weight(i)$. For every $weight(i)$, $weight(0) = 1$. $weight(i)$s for $i > 0$ are defined as follows.

- $weight(i) = \{1, 0.5, 0.2\}$ (fixed values)
- $weight(i) = \gamma^i$ (exponential function)
- $weight(i) = \delta - \epsilon i$ (linear function)

We evaluated with the fixed values first. Then we optimized the parameters γ, δ, and ϵ. As a result, we selected 0.7, 0.5, 0.05 for the parameters γ, δ, ϵ in the method-level and 0.2, 0.2, 0.02 in the file-level, respectively.

We used 0.9 for α, a parameter defined in the previous method (See Section 3.3).

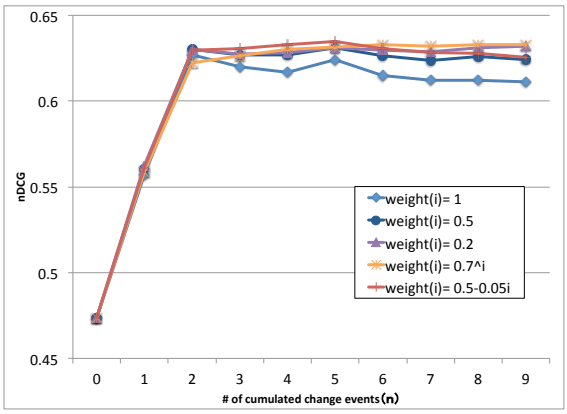

Figure 4. Method-level nDCG when changing n and $weight(i)$

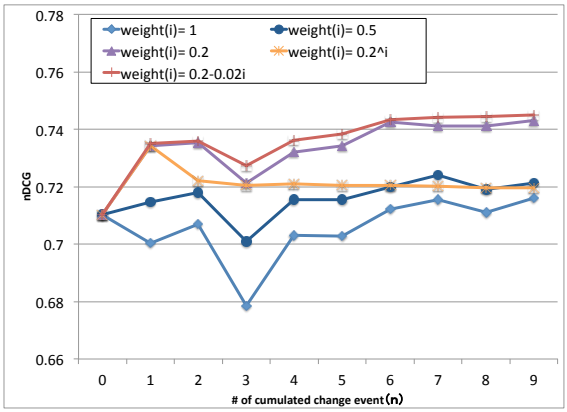

Figure 5. File-level nDCG when changing n and $weight(i)$

5.4 Result

Figures 4 and 5 show nDCGs when changing n and $weight(i)$.

In the method-level change recommendation, the parameters $n = 5, weight(i) = 0.5 - 0.05i$ marked the maximum nDCG (0.635), which is increased by 34.2% from the parameter $n = 0$ (nDCG $= 0.473$). The p-value of the Wilcoxon signed-rank test was 0.006, which confirms that the improvement of method-level recommendation was statistically significant under the confidence interval of 0.05.

On the other hand, improvement of the file-level change recommendation was not so much as method-level. In file-level, the parameters $n = 9, weight(i) = 0.2 - 0.02i$ marked the maximum nDCG (0.745), but it is increased by only 4.9% from the parameter $n = 0$. There are no parameters that produce significantly different nDCG from the nDCG of $n = 0$, which means that nDCG is not significantly increased or decreased.

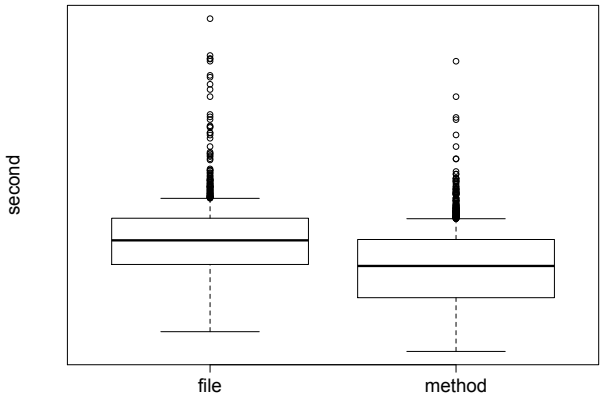

Figure 6. Time distribution of changing files and methods

5.5 Discussion

According to Figure 4, nDCG increased monotonically in the section of $0 \leq n \leq 2$ in the method-level with any $weight(i)$ functions. Therefore, cumulating two likelihoods to the latest likelihood improved the accuracy of the change recommendation in the method-level, regardless of $weight(i)$.

We used three constant weights and two decreasing weights and decreasing weights marked better nDCG than constant weights. Therefore, the older the likelihood is, it is necessary to decrease the weight.

In contrast, we could not find significant improvement in the file-level change recommendation. This is because methods are finer-grained than files and developers tend to switch the methods more frequently than the files. Figure 6 shows the box plot of the consumption time of change events in both granularity. The medians of consumption time were 96.6 and 33.6 seconds in the file-level and method-level recommendation, respectively. So the recommendation shown previously was still worthwhile after changing other methods so that cumulating likelihoods was effective in the method-level change recommendation.

We investigated why nDCG of our method was increased from the previous method in the method-level. We found three cases of interaction where the cumulative likelihood improved the accuracy of change recommendation.

Case 1: Change propagates two or more methods An interaction history of certain developer has the change events recorded as follows:

```
1  Fireball#move()
2  Sparkle#move()
3  Sparkle#Sparkle()
4  Fireball#move(float x, float y)
```

This sequence indicates that the developer changed `Sparkle` class after changing `Fireball#move()` method and then he returned to the `Fireball#move(float x, float y)`, which is located in the same file as the first change event. At this time, the recommendation of the previous method could predict the 4th method (`Fireball#move()`) to change after the 1st method (`Fireball#move(float x, float y)`) was changed and it could not predict other valuable changes. Since it does not keep the change recommendation after the next change happens, it could not predict the 4th change event after the 3rd change event happened.

Our proposed method overcame this limitation. It cumulated the likelihoods of recommendation after the 1st, 2nd, and 3rd change events and predict the 4th change event with this cumulated likelihood. It enabled to predict the 4th change event as the most likely to change after the 3rd change event.

Case 2: Changes of 2 or more methods propagate the same method An interaction history of certain developer has the change events recorded as follows:

```
1 Fireball#move()
2 Fireball#move(float x,float y)
3 Fireball#Fireball(LevelScene level, float x, float
      y, int facing)
```

The previous method predicted the 3rd change event, `Fireball#Fireball(LevelScene level, float x, float y, int facing)`, as the *3rd most likely method to change* after the 2nd change event, `Fireball#move(float x, float y)`. It also predicted the 3rd change event as the *3rd most likely method to change* even after the 1st change event, `Fireball#move()`.

Our proposed method was effective in this case because it cumulated the likelihoods of 1st and 2nd change events and it can predict the 3rd change event in higher likelihood than only the likelihood of 2nd change event.

Case 3: Changes of method Signatures An interaction history of certain developer has the change events recorded as follows:

```
1 FireEnemy#FireEnemy(LevelScene level; int x, int y)
2 FireEnemy#FireEnemy(LevelScene level; int x, int y,
       int a)
3 FireEnemy#FireEnemy(LevelScene level; int x, int y,
       int a, int b)
4 FireEnemy#move()
```

These were recorded because the arguments of the method `FireEnemy#fireEnemy()` have been changed twice and the previous method and our proposed method dealt with these three change event separately. Our recommendation engine could not track the signature changes and could not treat these methods as the same one.

The previous method could not recommend the 4th change event, `FireEnemy#move()`, after the 3rd change event even though it could after the 1st change event. On the other hand, our proposed method could recommend the 4th change event after the 3rd change event because it cumulated the likelihoods from 1st to 3rd. Our proposed method improved the accuracy in this case compared with the previous method.

5.6 Threats to Validity

5.6.1 Internal Validity

The interaction histories we used in the case study were those collected without change guide to the participants of the experiment. If the participants use our change guide method during development, they can reach appropriate artifact to change faster and the interaction histories differ from those without the guide. We should perform further case study by collecting interaction history recorded with our change guide method.

The participants probably performed the tasks with mistakenly interacting to the artifacts. We could not filter out such interaction events in the interaction histories. We dealt with the interaction events without distinguishing the proper interaction and the wrong interaction.

The tool, PLOG, can not track the signature changes and artifact deletions. Therefore, if the developer changes the signatures or deletes artifacts, our recommendation engine suggests the artifacts that have already been changed or deleted, which causes decreasing of the accuracy. And it also causes increasing of the accuracy as described in Section 5.5, Case 3.

5.6.2 External Validity

We used the interaction histories of 15 developers in the same development projects. We have to gather various type of interaction histories to generalize our results.

Since our approach, a change guide method based on fine-grained interaction histories, was done for the first time, we compared our result with our previous study as the baseline. We might have to compare with other type methods.

5.6.3 Construct Validity

We used the non-parametric Wilcoxon signed-rank test to assess whether the nDCG of the change recommendation was improved using cumulative likelihood. This is appropriate because we have no assertion about the normality of the data.

6 Conclusion

Our previous method of change guide could not deal with the situation when a change propagates more than one code element. We defined the cumulative likelihood to

overcome this limitation. We performed case study using 15 participants of interaction history logs and revealed that the cumulative likelihood could improve the accuracy of the change guide in the method-level.

To apply our method to the actual development scene, we are now planning to implement a tool for suggesting the candidate to change next to the developer as an Eclipse plug-in. It can capture the developer's interaction history, learn it just-in-time, and calculate the candidates of next change every time the developer changes to the code, using our proposed method. It can also output the interaction history as a log file, which can be used in the further research.

Acknowledgment

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

References

[1] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining version histories to guide software changes," *Proc. ICSE*, vol. 31, no. 6, pp. 429–445, 2005.

[2] L. Hattori, G. dos Santos Jr, F. Cardoso, and M. Sampaio, "Mining software repositories for software change impact analysis: a case study," in *Proc. SBBD*, 2008, pp. 210–223.

[3] L. Briand, J. Wust, and H. Lounis, "Using coupling measurement for impact analysis in object-oriented systems," in *Proc. ICSM*, 1999, pp. 475–482.

[4] M. M. Geipel and F. Schweitzer, "Software change dynamics: Evidence from 35 java projects," in *Proc. ESEC/FSE*, 2009, pp. 269–272.

[5] 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.

[6] H. Gall, K. Hajek, and M. Jazayeri, "Detection of logical coupling based on product release history," in *Proc. ICSM*, 1998, pp. 190–198.

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

[8] W. Maalej, T. Fritz, and R. Robbes, "Collecting and processing interaction data for recommendation systems," in *Recommendation Systems in Software Engineering*, M. P. Robillard, W. Maalej, R. J. Walker, and T. Zimmermann, Eds. Springer Berlin Heidelberg, 2014, pp. 173–197.

[9] T. Kobayashi, N. Kato, and K. Agusa, "Interaction histories mining for software change guide," in *Proc. RSSE*, 2012, pp. 73–77.

[10] W. C. Hill, J. D. Hollan, D. Wroblewski, and T. McCandless, "Edit wear and read wear," in *Proc. CHI*, 1992, pp. 3–9.

[11] M. Kersten and G. C. Murphy, "Mylar: a degree-of-interest model for ides," in *Proc. AOSD*, 2005, pp. 159–168.

[12] Y. Yoon and B. A. Myers, "Capturing and analyzing low-level events from the code editor," in *Proc. PLATEAU*, 2011, pp. 25–30.

[13] 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 *Proc. ECOOP*, 2012, pp. 79–103.

[14] S. Negara, M. Codoban, D. Dig, and R. E. Johnson, "Mining fine-grained code changes to detect unknown change patterns," in *Proc. ICSE*, 2014, pp. 803–813.

[15] R. Minelli, A. Mocci, M. Lanza, and T. Kobayashi, "Quantifying program comprehension with interaction data," in *Proc. QSIC*, 2014, pp. 276 – 285.

[16] Z. Gu, D. Schleck, E. T. Barr, and Z. Su, "Capturing and exploiting ide interactions," in *Proc. Onward!*, 2014, pp. 83–94.

[17] L. Zou, M. Godfrey, and A. Hassan, "Detecting interaction coupling from task interaction histories," in *Proc. ICPC*, 2007, pp. 135–144.

[18] R. Robbes, D. Pollet, and M. Lanza, "Logical coupling based on fine-grained change information," in *Proceedings of the 15th Working Conference on Reverse Engineering*, 2008, pp. 42–46.

[19] ——, "Replaying ide interactions to evaluate and improve change prediction approaches," in *Proc. MSR*, 2010, pp. 161–170.

[20] W. Maalej and A. Sahm, "Assisting engineers in switching artifacts by using task semantic and interaction history," in *Proc. RSSE*, 2010, pp. 59–63.

[21] T. Roehm and W. Maalej, "Automatically detecting developer activities and problems in software development work," in *Proc. ICSE*, 2012, pp. 1261–1264.

[22] K. Järvelin and J. Kekäläinen, "Cumulated gain-based evaluation of ir techniques," *ACM Trans. Inf. Syst.*, vol. 20, no. 4, pp. 422–446, Oct. 2002.