Abstract—This paper presents a prediction model for change propagation based on the developers’ interaction history. Since artifacts have internal and external dependencies, a change will cause some changes on related artifacts. In order to guide change operations in software development, our proposed method generates a change guide graph by mining developers’ interaction histories which consist of write and read accesses to artifacts. Using a change guide graph, we can guide change using the context of previous changes. To evaluate proposed change guide method, we perform a case study with an open-source software. We show that the context information is effective for file level and method level change predictions.

I. INTRODUCTION

Impact analysis is one of the key tasks on software maintenance. Since there are complex dependencies among modules and/or other artifacts such as configuration files, test codes and documents, a change often propagates some new changes on related artifacts. If a developer fails to track dependencies and to change appropriately, some bugs will be introduced. To address this issue, some impact analysis methods based on static program analysis have been proposed [1]. However, the results of these approaches typically are difficult to scale out and include false-positive answers since these approaches treat fine-grained but partially abstracted dependencies on program codes [2].

As other trends of supporting maintenance tasks, change guide methods based on analysis of developers’ activity have been proposed. Developer on test and maintenance spend a great deal of time in his analysis to find relevant problems solved past developments [3]. Gall et al. introduced a concept of logical coupling, which is a relation between modules modified same timing frequently [4]. This concept has been used in many research proposals [5]–[7] and change recommendation systems such as eRose [8], sqminer [9], ChangeCommander [10], [11] which extract the relation from developers’ activity histories by using data mining technique. When most developers modify codes in maintenance tasks appropriately, these methods provide more accurate suggestions.

However, the change suggestions still include many false-positive answers because these methods extract developers’ activity from a software repository. Since activity histories in a software repository record only on commit timing, we cannot treat information of other activities between commits such as referring files, order of writing files.

In this paper, we focus to analyze interaction history [12] which is detailed developer’s activity history. Several researches related to interaction history has been proposed [13]–[17]. We propose new change guide method by analyzing ‘missing’ activities between commits. We capture not only committing but also referring artifacts with interaction histories.

In our proposed method, we store interaction histories by capturing operation of IDE and extract a sequence of changes with context information. We then generate a change guide graph by analyzing sequences.

We evaluate our proposed method with a case study using a open source software. Experimental results show the context information extracted from interaction histories increase the precision of change guides.

II. RELATED WORK

Interaction history [12] is a detailed history of developers’ activity includes not only modification but also reference and search. Zou et al. [13] proposed a method to classify maintenance tasks based on “interaction coupling” which is a relationship between artifacts frequently refereed. Mylar [14], which is a system to support comprehension, identify artifacts related to current tasks by analyzing interaction histories. Parmin et al. [15] and Singer et al. [16] also have proposed interaction history based method to identify related artifacts and suggest artifacts which developer should refer to.

Goal of most existing research using interaction histories are limited to support comprehension. Robbes et al. [17] have proposed a fine-grained logical coupling based on interaction history. However, they focused on history of modification only.

III. SOFTWARE CHANGE GUIDE WITH INTERACTION HISTORIES

We propose a change guide method based on a change guide graph (CGG) which is a prediction model for change
propagation generated by mining interaction histories. We first capture and store developers’ interaction histories and generate a CGG from interaction histories. By using CGG, we can guide changes using the context of previous changes which is not considered in existing studies.

A. Capture Interaction Histories

We developed an Eclipse plug-in “plog” for capturing interaction histories. The plog can capture developers’ interactions with software artifacts by listening to various IDE events. The plog can capture two grains histories which are a file level and a method (in only Java programs) level. In our approach, we treat histories which consist of read and write accesses to artifacts.

Figure 1 illustrates the difference between a typical history extracted from repository and our interaction history.

To eliminate noise from histories, we use preprocessed interaction histories. We define minimum access time $T_{min}$ and maximum access time $T_{max}$ for cleansing. If an access time is shorter than $T_{min}$, we consider the activities as miss-operation and removed the entry from access logs. $T_{max}$ is used to treat long activities such that a developer starts other tasks or leaves his seat with files open in a editor. We shorten these long activities to $T_{max}$.

B. Generating a Change Guide Graph (CGG)

1) Generating Attributed Change Sequence: We generate a CGG by mining to developers’ interaction histories. We first extract “attributed change sequences (ACS)” from interaction histories as base information for CGG.

In our approach, we define a “change” as continuous write accesses to a same artifact. Suppose a developer writes access to an artifact $A$, views an artifact $B$, then writes to a $A$ again. We consider two writing to $A$ as a change to $A$ and store a history as “a developer views $B$ while changing to $A$”.

Each of entry in ACS is a change with attribute which expresses a context to a change. The attribute is a vector whose dimension is a number of artifacts. Each value of components represent whether the context to a change contains each artifact.

Since we consider that the artifacts changed before a change affects the change, a context to a change consists of referred artifacts for the change and $n$ artifacts changed before the change. In this paper, we use $n=1$, which mean we treat the last modified artifacts before the change as context only. Using multiple changed artifacts as the context is future work.

All artifacts contained in a context will not affect a change evenly. In our method, we determine each value of components of an attribute vector considering an impact for a change. The impact of previous changes for a current change is different from other context information. We set context value for previous changes by using constant $X$.

We assume that effects of following artifacts is strong:

- Artifacts which are accessed close to a change.
- Artifacts which are accessed long time.

When developers develop a new feature by reference other artifacts, developers will switch his focus between changed artifacts and referenced artifacts. In this situation, developers will reference artifacts related to the change strongly at close to the change access.

To express impact based on our assumption, we use a value integrated a weight function $f(t)$ over the period of access time.

Fig. 2 shows a calculation example of weight of an artifact $A$ for change of an artifact $B$. $A$ was accessed three times during change of $B$. In our approach, we use $f(t) = e^{a(t-t_{base})}$, $a = 0.01$. This function returns 1 for closest position and about 0.01 for 40 minutes before.

2) Generating CGG: We generate a CGG from ACS. The CGG is a directed graph of which edges represent sequences of changes. Figure 3 illustrates an example of CGG. Each
of edges has one or more attributes. We generate the graph by following steps:

1) Pick up consecutive two changes from a ACS.
2) Find an edge from an artifact changed before to an artifact changed later.
3) We add an attribute vector of a previous change to a set of attribute vectors of the edge. If the edge does not exist, we generate an edge from previous change to next change.

Attribute vectors of an edge represent a history of change propagation. An attribute is contained a context information, and it is added singly by every change in that order. We can store change propagation histories in CGG by repetitiveness these steps.

C. Change Guide by Using Change Guide Graph

In our approach, we guide next changes by using CGG. A likelihood for next change is calculated with a change and attributes which are matched with the context of the change. We calculate a likelihood \( L_e(w) \) of each edge \( e \in E' \) for change \( w \) by a following expression.

\[
L_e(w) = \sum_{v \in A_e} \{(1 - \alpha) + \alpha (\text{context}(c_w, v))\}
\]

Where \( w \) is a new change triggered guide process and \( c_w \) is an attribute of \( w \). \( E' \) is a set of edges directed outward from a changed node in the CGG. \( A_e \) is a set of attributes of each of edges \( e \in E' \). The \text{context} represents strength of matching context of changes. Our current implementation of \text{context} is a dot product of two vectors of context information.

The \( \alpha \) is a parameter which represents the degree of concerning a context information. If \( \alpha = 0 \), the \( L \) is independent of \text{context}. The \( L \) of an edge is equal to the number of attributes of the edge, the value is equal to change frequency in that order. If \( \alpha = 1 \), the \( L \) is dependent on a context information only. In this case, the \( L \) is zero if there is no common context even if there are a lot of attributes.

IV. CASE STUDY

A. Experimental Setup

To evaluate our approach, we conducted a case study. Six students in our laboratory volunteered to participate in this study. As a target software, we used the JPacman\(^1\): a open source game implemented by Java. It has 54 classes and 6KLOC. Participants expanded the functions of JPacman with IDE. We captured their interaction histories by plog. Outlines of the feature additions are shown in the following.

- **Feature 1: Bomb**

  Pacman can drop a bomb with holding B button down. If pacman touch a bomb, player lost a life. If a ghost touch a bomb, the ghost become a death state.

- **Feature 2: Pitfall**

  Pacman can generate a pitfall with holding H button down. If pacman touch a pit, pacman stay in there for three seconds. If a ghost touch a pit, the ghost stay in there for five seconds.

We made check sheets instead of using test code. Participants verified their extension with using the sheets. Table I shows the number of logs which was captured during their developments. In the Table I, “r” and “w” represent the number of read and write accesses to artifacts, “c” is the number of change which has been defined in III-B1.

We evaluated about the following items using captured interaction histories:

- **Evaluation 1**: Effectiveness of file level change guide based on our method.
- **Evaluation 2**: Effectiveness of method level change guide based on our method.
- **Evaluation 3**: Relationship between accuracy and likelihood

We selected training data and test data from captured access logs. In this evaluation, we set all students’ access logs of feature 1 as training data, and set each students’ access logs of feature 2 as test data.

We generated a CGG from training data, and generated a change sequence from test data. Using the CGG we calculated which artifacts were required to change next to a change in change sequence. We defined an artifact which had been changed next in history as correct one. We evaluated the rank of a correct answer.

\(^1\)http://code.google.com/p/jpacman/
As evaluation metrics, we used the mean reciprocal rank (MRR) defined as follows:

\[
MRR = \frac{\sum_{i=1}^{n} 1/rank_i}{n}
\]

where \(rank_i\) is the rank of best correct answer for \(i\)th query.

Please note that \(rank_i\) is 0 when a result of the query includes no correct answer. The higher MRR, the more effective we guide change. If there are tie ranks, we treat their rank as the lower one. For example, likelihood values for four artifacts is \(\{50, 20, 20, 10\}\), we define ranks as \(\{1, 3, 3, 4\}\).

We set \(T_{\text{min}} = 1[s]\) and \(T_{\text{max}} = 1800[s]\) for data cleansing and \(X\), which is a weight of previous change, is 0.5. We conducted case studies, changing \(\alpha\) from 0 to 1. Note that the setting of \(\alpha = 0\) is a baseline approach which just use change frequency of files after a change. The approach does not consider context information.

B. Evaluation 1: File Level Change Guide

Figure 4 shows evaluation results of the guiding change which focused on files. The horizontal axis shows the value of \(\alpha\) in an expression 1, the vertical axis shows the value of MRR in each \(\alpha\).

In result of \(D\) and \(E\)'s data, MRR increased as \(\alpha\) increased. This result shows that the context information is effective for change guide which focus on files.

In result of \(A\)'s data, MRR varied only slightly as \(\alpha\) increased. We could not guide change in the most cases for \(A\). The reason is that there are no appropriate edge in a CGG because \(A\) changed artifacts which nobody had changed.

C. Evaluation 2: Method Level Change Guide

Figure 5 shows result of MRR for method level change guide. The context information has positive effects until \(\alpha = 0.9\). However, all values of MRR are under 0.12 and MRR of \(\alpha = 1\) is lower than one without context \((\alpha = 0)\). This is because the amount of training data is not enough to construct a usable change guide graph. Most of the suggestions by using the change guide graph match few context information and likelihood of these suggestions are low because we use detailed interaction histories. If two developer access different methods in the same class, these interactions treat same activity in the class level interaction, however, these should be distinguished in the method level one.

D. Evaluation 3: Relationship between Accuracy and Likelihood

Our propose method suggests every artifact connected with edges \(e\) even if the \(L_e\) value is low. Lower \(L_e(w)\) value means that the context of modification \(c_w\) and past modifications for the artifact are not so similar.

To investigate a relationship between accuracy of suggestion and likelihood defined in III-C, we measure accuracy when we restrict suggestion by using a threshold for likelihood \(T_{L}\).

We calculate the rank of a suggestion by using candidates \(\alpha\) and \(T_{L}\). This means that we use highly confident candidate only. Table II shows comparison of ranks with \(\alpha = 0\) and with \(\alpha = 1\) varying \(T_{L}\). We can find that suggestions with high likelihood \(L_e(w)\) can improve the rank. On the other hand, ones with low \(L_e(w)\) is negative affection for ranking. There are 15 suggestions of which

\[\begin{array}{|c|c|c|c|}
\hline
\text{Threshold of } & \text{improved} & \text{degraded} & \text{not changed} \\
\text{Likelihood } (T_{L}) & 9 & 16 & 8 \\
\hline
>0.0 & 9 & 7 & 4 \\
>0.1 & 9 & 5 & 1 \\
>0.2 & 9 & 5 & 1 \\
>0.3 & 8 & 1 & 0 \\
\hline
\end{array}\]
values is under 0.3. These suggestions have lower rank than without context information ($\alpha - 0$). That is to say, likelihood $L_v(w)$ is an adequate indicator for suggestion. If we have enough history to build CGG and define the $T_L$, we can guide changes appropriately.

V. CONCLUSION AND FUTURE WORK

In this paper, we have proposed a method to guide changes on maintenance tasks. Our proposed method is based on a change guide graph which mined from developers’ interaction histories. We treat not only logical coupling but also sequences of reference and modification for artifacts as context of a change. Experimental results show the context information can improve precision of file level change predictions and our method also can support method level change predictions when we have enough histories. We currently have plans to organize the additional experiment on more interaction histories and extend our change guide method to considering more past modification in context.

ACKNOWLEDGMENT

We would like to thank Dr. Shinpei Hayashi and Dr. Noritoshi Atsumi for their invaluable feedback. This work is partially supported by a Grant-in-Aid for Scientific Research of MEXT Japan (#20300009, #21700027, #21013017).

REFERENCES


