Predicting the Next Change at the Fine-Grained Level

Hiroaki Murakami, Keisuke Hotta, Yoshiki Higo, and Shinji Kusumoto
Osaka University
1-5 Yamadaoka, Suita, Osaka, Japan
{h-murakm,k-hotta,higo,kusumoto}@ist.osaka-u.ac.jp

ABSTRACT
Changing source code is not an easy task. Developers occasionally change source code incorrectly or overlook code fragments that need to be changed. Such mistakes entail additional cost in having to reedit the source code correctly, and repeated changes themselves can be a hazard to software quality. We are conducting research into realizing automated code changing as a countermeasure for human errors. As the first step of this research, in this paper we propose a technique to predict what kinds of program elements will be deleted and added in the next change to Java methods. We also evaluate two types of prediction using the proposed technique. One is predicting all types of program elements. The other is predicting only program elements that are statements or larger elements. The precision of the two types of prediction was experimentally found to be 54%-96% and 73%-99%, respectively.

Categories and Subject Descriptors
D.2.2 [Design Tools and Techniques]: Computer-aided software engineering; D.2.7 [Distribution, Maintenance, and Enhancement]: Version control

General Terms
Management

Keywords

1. INTRODUCTION
Changing source code is the most costly activity in software maintenance [21], and it is not an easy task. Developers occasionally change source code incorrectly or forget to change code fragments. Such mistakes require not only additional human resources to reedit the code correctly, but also these mistakes in and of themselves are not good for software quality [16]. Consequently, facilitating source code changes by introducing automatic operations would be very beneficial.

A variety of techniques for facilitating source code changes have been proposed. For example, there are techniques to predict modules in which changes frequently occur by constructing prediction models [2, 3, 15]. There is also a technique that is tailored to interfaces, which are generally stable components in software systems [19]. Tsantalis et al. proposed a technique to calculate the probability that each class will be changed in a future version [23]. By using these techniques, development projects are able to spend human resources on predicted fault-prone modules. For example, if we obtain fault-prone modules, we can focus on reviews for them.

Another hot research topic is identifying pairs or sets of modules that need to be changed together [10, 11, 18, 22, 27, 28]. By using these techniques, developers are able to know which other modules need to be changed together, after they identify some modules that need to be changed. Such a support mechanism prevents developers from forgetting to change code fragments in a given task.

Facilitating bug fixes is a particularly hot research topic. There are various techniques such as prioritizing open bugs for fixing [1] identifying code fragments that cause given bugs [9, 20], and validating that bug fixes have been performed correctly [26].

There are many approaches to fixing bugs automatically (the generation of patches for bug fixes). Perkins et al. developed ClearView, which analyzes a target program dynamically and generates patches for satisfying invariants in the program [17]. Wei et al. proposed a technique that generates patches that satisfy not only invariants but also preconditions and postconditions [24]. Wei’s technique is able to fix more bugs than Perkins’s technique because Wei’s technique uses preconditions and postconditions. However, programs must have descriptions of invariants, preconditions, and postconditions in their source files to be analyzed using Wei’s technique. In Perkins’s technique, invariants are automatically identified by using a tool Daikon; thus, programs do not need to have descriptions of invariants in their source files. Jin et al. proposed a technique to fix bugs that violate the atomicity of parallel procedures [8]. Weimer et al. developed a tool called GenProg that uses genetic programming to generate patches for fixing bugs [14, 25], and they reported that GenProg was able to fix 55 out of 105 bugs automatically [13].
Source code changes are performed not only to fix bugs but also to add new functionalities and change/enhance existing functions. Of course, we need to take into account requirements for changing/enhancing existing functions. On the other hand, historical approaches to guess the next change seem promising because “programs that people write are mostly simple and somewhat repetitive” [7].

In this research, we firstly investigate the size of source code changes during software evolution. As a result, we found that most of the changes are small and simple. Then, we develop a new technique to predict how a given method will be modified in the next change. Our technique can predict any size of changes in source code, but in particular its works well at small changes.

Our technique is beneficial to individual developers when he/she needs to change source code to complete a given task. Our technique facilitates source code changes by showing (predicting) the source code of the next version. If the predicted source code satisfies his/her requirements, he/she can obtain a patch to change the current source code to the next version. By using the proposed technique,

- developers do not need to consider how to change code fragments, and
- they do not need to change source code manually.

This paper makes the following contributions:

- We propose a novel technique that predicate the next changes. Currently, our technique predicts which types of program elements will be added and deleted in the next change.
- We evaluate our technique by conducting experiments on three open-source software systems. We evaluate two types of prediction: one is predicting all types of program elements; the other is predicting only program elements that are statements or larger elements. The precision of the two types of prediction are 54%-96% and 73%-99%, respectively.

The remainder of this paper is organized as follows: Section 2 introduces related works and discusses the differences between them and our technique. Section 3 provides an overview of our technique, and Section 4 describes how to build prediction models. Section 5 shows how to implement our technique. Section 6 describes design of experiment, and Section 7 reports experimental result in detail. Section 8 discusses the experimental result by using examples of source code. Section 9 describes threats to validity. Section 10 concludes the paper and refers to the future work.

2. RELATED WORK

There are a number of approaches that use source code metrics to train change-prone or fault-prone prediction models, which can lead developers to such modules in a software system [2, 3, 15, 19]. Although the experimental results of these approaches are promising, they do not provide insights into the details of changes.

Giger et al. explored prediction models for whether a source file will be affected by certain types of changes such as condition changes, interface modifications, inserts or deletions of methods and attributes, or other kinds of statement changes [5]. Their models output a list of potentially change-prone files ranked according to their change-proneness overall and per change type category. The purpose of their research and ours is the same, which is to provide insights into the details of the changes. However, the methods used for gaining such insights are different. Giger’s approach predicts the category of code changes, whereas our current approach predicts what types of program elements are deleted and added.

Goues et al. proposed a technique to repair programs automatically [14, 25]. Their tool, GenProg, uses genetic programming to repair a wide range of defect types in C software (e.g., infinite loops, buffer overflows, segmentation faults, and integer overflows). GenProg searches for a repair method that retains the required functionality by generating variant versions of the program through computational analogs of biological processes. They succeeded in creating source code itself, which is a repaired version. Kim et al. also proposed learning to fix patterns from human-written patches to improve the quality of generated patches because GenProg sometimes generates nonsensical patches due to the randomness of its mutation operations [12]. Their techniques are tailored to repairing bugs. It is difficult to apply their technique to other kinds of changes such as adding functionalities or enhancing functions because it requires test cases that a given bug passes and fails. On the other hand, the target of our approach is not only bug fixes but any kinds of changes in source code such as functional addition/enhancement or refactoring.

3. OVERVIEW OF OUR TECHNIQUE

This paper proposes a technique to predict how to change a given Java method at the program element level. For instance, it tells us “one if statement, two identifiers, and one assign expression will be added by the next change.” It learns past changes from a historical code repository, and builds prediction models.

Users of the proposed technique should input a Java method that needs a change. The proposed technique then predicts the next change on the given method.

This section introduces terms that are used in this paper followed by an overview of the procedure of the proposed technique. The details of the procedure are described in the following section.

3.1 Terms

Program Element

This paper uses the nodes of an abstract syntax tree (AST) as program elements. Each type of node is treated as a type of program element. We borrow the definitions of AST nodes from Java Development Tools (JDT). JDT provides us with a function to build ASTs from given Java source files. It defines 83 types of AST nodes including three related to comments1. We use 80 types of program elements because we are not interested in comments. This paper uses symbols \( A_0, \ldots, A_79 \) to represent each type of program element.

1The types of AST nodes in JDT and this study are defined in the class org.eclipse.jdt.core.dom.ASTNode. A list of them is provided in the Éclipse documentation. The current status is available from “http://help.eclipse.org/kepler/index.jsp.” Note that the types of AST nodes might change due to an update of Java or JDT.
The proposed technique adopts a two-stage prediction as described below.

**Prediction for Filtering**: Predicting whether the next change that a given method undergoes will be *small*. This stage filters out methods that are predicted as undergoing *large* changes in their next revisions.

**Prediction of Changes**: Predicting how many program elements will be added/deleted in the next change for a given method. More concretely, the proposed technique predicts the next PE-Vector of a given method $m$. For instance, suppose $m$ becomes $m'$ in the next change, the proposed technique predicts $v_{m}′$ based on $v_m$. We can know how many program elements will be added/deleted in the next change from the differences between $v_m$ and $v_m′$. This stage targets only methods that are predicted as undergoing *small* changes in the first stage of prediction.

The noteworthy point of the proposed technique is that it excludes methods from prediction targets if they are predicted as undergoing *large* changes in the next revision. The rationale behind this filtering is that it is difficult to predict large changes accurately. It is necessary for the proposed technique to make precise predictions because of our eventual goal, which is evolve software systems automatically. The proposed technique will be the first step in the evolution of automated software. Hence, the accuracy of a prediction made with the proposed technique is important for the final goal. That is, if an incorrect prediction were made, this would lead to an incorrect evolution. We believe that an incorrect evolution must be avoided, even although we will lose some candidates that might be able to be evolved automatically. Therefore, we have decided to ignore large changes currently by means of such filtering. However, we will improve our technique to be able to predict large changes correctly as a future task.

The above begs the question, “What is a *small* change?” This study considers a change to be *small* if the number of types of program elements that were added/deleted by the change is less than a given threshold.

Suppose a method $m$ was changed and became $m'$, and let $\vec{v} = (x_0, \ldots, x_79)$ be a difference vector from $m$ to $m'$ ($\vec{v} = v_{m}′ - v_m$). The change to $m$ is regarded as *small* if the following formula holds, otherwise it is regarded as *large*.

$$|\text{changed}(\vec{v})| \leq \text{threshold}$$  \hfill (1)

where,

$$\text{changed}(\vec{v}) = \{ i \in 0 \ldots 79 \mid x_i \neq 0 \}$$  \hfill (2)

There is a reason we use the number of types of program elements instead of code churn, which is a commonly metric to measure the size of a change. The reason is that code churn is unsuitable for this study because of the granularity of the prediction. The proposed technique estimates the number of additions/deletions of program elements. In other words, it can predict only the type of the next change, not the content of the change. More concretely, the proposed method can tell “a *if* statement will be added,” but it cannot tell the predicate and the body of the *if* statement. Code churn is measured based on the contents of changes, e.g., the

---

**PE-Vector**

The proposed technique treats each method as a vector named a Program Element Vector (PE-Vector). PE-Vector is a numerical vector that has 80 dimensions, and each element of a PE-Vector represents the number of program elements in the method. This paper describes the PE-Vector of a method $m$ as $v_m = (x_0, \ldots, x_{79})$. Here, $x_i$ in $v_m$ denotes the number of program elements $A_i$ in the method $m$.

Figure 1 shows an example of a PE-Vector. Figure 1(b) shows an AST for the method shown in Fig. 1(a), and the vector shown in Fig. 1(c) is the PE-Vector created from the method. The vector has 80 dimensions; however, we omitted the attributes from a figure whose value is 0, except for the head and the tail of the vector, due to space limitations.

**3.2 Overview of the Procedure**

Figure 2 provides an overview of the procedure of the proposed technique and a prediction made using it. The proposed technique requires a historical code repository as its input, and builds prediction models.

As the first step in prediction, the technique generates a training set from a given historical code repository. The training set has information about past changes in the historical repository. An element of the training set is a pair of PE-Vectors $v_m$ and $v_{m'}$, where $m$ was changed and became $m'$.

Note that the PE-Vectors before and after a change might be the same vector. This means that the change did not change the number of program elements. For instance, this case occurs when the change is renaming of local variables.

With the training set, the proposed technique builds prediction models. The models tell us how many program elements will be added/deleted in the next change for a given method.

The proposed technique will be the first step in the evolution of automated software. Hence, the accuracy of a prediction made with the proposed technique is important for the final goal. That is, if an incorrect prediction were made, this would lead to an incorrect evolution. We believe that an incorrect evolution must be avoided, even although we will lose some candidates that might be able to be evolved automatically. Therefore, we have decided to ignore large changes currently by means of such filtering. However, we will improve our technique to be able to predict large changes correctly as a future task.

The above begs the question, “What is a *small* change?” This study considers a change to be *small* if the number of types of program elements that were added/deleted by the change is less than a given threshold.

Suppose a method $m$ was changed and became $m'$, and let $\vec{v} = (x_0, \ldots, x_79)$ be a difference vector from $m$ to $m'$ ($\vec{v} = v_{m}′ - v_m$). The change to $m$ is regarded as *small* if the following formula holds, otherwise it is regarded as *large*.

$$|\text{changed}(\vec{v})| \leq \text{threshold}$$  \hfill (1)

where,

$$\text{changed}(\vec{v}) = \{ i \in 0 \ldots 79 \mid x_i \neq 0 \}$$  \hfill (2)

There is a reason we use the number of types of program elements instead of code churn, which is a commonly metric to measure the size of a change. The reason is that code churn is unsuitable for this study because of the granularity of the prediction. The proposed technique estimates the number of additions/deletions of program elements. In other words, it can predict only the type of the next change, not the content of the change. More concretely, the proposed method can tell “a *if* statement will be added,” but it cannot tell the predicate and the body of the *if* statement. Code churn is measured based on the contents of changes, e.g., the
number of changed lines or tokens of a change. Therefore, it cannot be measured without the content of a change. We believe that we should measure the size of a change without something we cannot predict. In this case, we should use the types of changes instead of their contents.

As mentioned above, the PE-Vectors before and after a change might be the same vector. Hence, it is possible that the proposed technique reports the same PE-Vector of a given method as a prediction result. Such a prediction result indicates that the method needs a change that does not change the number of program elements such as renaming of variables. This result does not indicate the method does not need any changes. This is because the proposed technique supposes that users of it input methods that should not need any changes. This is because the proposed technique reports the same PE-Vector of a method as a prediction result. Such a prediction result indicates that the method needs a change that does not change the number of program elements such as renaming of variables. This result does not indicate the method does not need any changes. This is because the proposed technique supposes that users of it input methods that should not need any changes.

3.3 Restrictions

The proposed technique currently has the following restrictions.

- The target of prediction is limited to code inside Java methods. Programming languages other than Java, or code outside Java methods are outside the scope of the proposed technique.
- The proposed technique cannot predict changes that are simultaneously performed on multiple methods. It just predicts the next version of each method.
- The proposed technique cannot generate source code. It just predicts what kind of program elements are added/deleted in the next change.

4. PREDICTION MODELS

This section describes how to build prediction models using the training set. As mentioned above, the proposed technique adopts a two-stage prediction. The first stage makes predictions for excluding methods from the prediction target. As mentioned above, the PE-Vectors before and after a change might be the same vector. Hence, it is possible that the proposed technique reports the same PE-Vector of a given method as a prediction result. Such a prediction result indicates that the method needs a change that does not change the number of program elements such as renaming of variables. This result does not indicate the method does not need any changes. This is because the proposed technique supposes that users of it input methods that should not need any changes.

As mentioned above, the PE-Vectors before and after a change might be the same vector. Hence, it is possible that the proposed technique reports the same PE-Vector of a given method as a prediction result. Such a prediction result indicates that the method needs a change that does not change the number of program elements such as renaming of variables. This result does not indicate the method does not need any changes. This is because the proposed technique supposes that users of it input methods that should not need any changes.

4.2 Prediction of Changes

The second stage of prediction uses linear regression analysis to predict the next PE-Vector of a given PE-Vector. This stage builds a regression function for each of the attributes in PE-Vector. In other words, it makes predictions on each program element independently. Hence, this stage generates the same number of regression functions to the dimension number of PE-Vector, which is 80.

Let \( x_0, \ldots, x_{79} \) be the target of prediction, and \( y_0, \ldots, y_{79} \) be a PE-Vector created by the prediction for \( v_m \). The proposed technique builds a regression function for each of the attributes \( a_0, \ldots, a_{79} \). Equation (3) describes the form of each regression function.

\[
\hat{y}_i = \sum_{j=0}^{79} \beta_{ij} x_j + \epsilon_i
\]
Equation (3) states that the value of an attribute \( A_i \) is estimated with those of all the attributes \( A_0, \ldots, A_{29} \). However, we should not use all the attributes as explanatory variables because there are correlations among some of the attributes. Hence, we have to select which variables to be used as explanatory variables to avoid multicollinearity. Therefore, \( \beta_j \) will have a nonzero value if the attribute \( A_j \) is selected as an explanatory variable, otherwise zero.

Note that each of the regression functions should have different explanatory variables because they are built independently. In other words, it is possible that \( x_j \) is used as an explanatory variable in the regression function for \( y_i \), but is not used in the one for \( y_k \).

5. IMPLEMENTATION

We have implemented the proposed technique with Java and R\(^2\). We assembled a tool to create a PE-Vector from the source code of a given method and to construct a training set from the given historical code repository in Java, while we entrusted the static analysis to build prediction models to the R functions.

Our implementation can handle software systems written in Java and managed using Subversion. We targeted Subversion repositories in this study, but it is easy to handle other control systems including Git and CVS. To extend our implementation to be able to handle other programming languages, we need to define the program elements for the languages. However, it is not difficult to extend our implementation for other languages once the definitions of the program elements are fixed.

To build the training set, it is necessary to detect which methods were changed in each of the past commits and how they were changed. We applied the clone tracking technique developed by our research group to this task [6]. This technique was originally used to mapping code clones between two consecutive revisions, but it can be used to map methods between two consecutive revisions. It is based on CRD [4], which is a text representation of the location of a given code fragment. The clone mapping technique links two code fragments between two consecutive revisions based on the similarity of their CRDs, and makes links of code clones between the two revisions based on the links of code fragments. The technique is well suited to handle file renaming and file moving because it uses the similarity of CRDs, not perfect matching of them. As well as the clone mapping technique, the proposed technique in this study developed CRDs from every method and makes links of methods between two consecutive revisions based on the similarity of their CRDs.

We used the \( \text{knn} \) function of R to predict small changes, and the \( \text{lm} \) function to perform the linear regression analysis. In addition, we adopted the method of increasing and decreasing the variables of the \text{step} function to select explanatory variables. The \text{step} function makes prediction models that have the lowest AIC, which contributes to avoid the problems of overfitting and multi-collinearity.

6. DESIGN OF EXPERIMENT

The purpose of the experiment conducted in this paper is to investigate to what extent the proposed technique can predict the number of added/deleted program elements in the next change correctly. In this experiment, we investigated two kinds types of prediction: one uses all the elements in PE-Vector; while the other uses only the statement elements. The statement elements are the elements that represent statements or large elements in AST provided by JDT. Figure 4 shows an example of statement elements, where the statement elements are the white nodes. By using only the statement elements, the number of inserted/deleted statements can be predicted. The reason we investigate prediction using only statement elements is to avoid overtraining by the elements except for the statements when we want to know only how will statements change in the next change. In total, the number of elements in PE-Vector is 80, 40 of which are statement elements.

To achieve the purpose of the experiment, we investigated the following three research questions.

\textbf{RQ1:} How accurately does the proposed technique predict whether the next change will be \textit{small} or not?

\textbf{RQ2:} How accurately does the proposed technique predict the next change when all the elements are used in PE-Vector?

\textbf{RQ3:} How accurately does the proposed technique predict the next change when only the statement elements are used in PE-Vector?

First, we conducted a preliminary experiment to decide the \textit{threshold} in Eq. (1) for the main experiment, and then we answered the research questions by conducting the main experiment.

The target software systems are \textit{OpenYMSG}, \textit{ArgoUML} and \textit{Ant}. \textit{OpenYMSG} is used for the preliminary experiment, and \textit{ArgoUML} and \textit{Ant} are used for the main experiment. These systems are written in Java and managed using Subversion. Table 1 shows the details of the systems.

In the preliminary experiment, we applied the proposed technique to \textit{OpenYMSG}, which is the smallest target. The reason we chose \textit{OpenYMSG} is because we need to investigate the various results yielded by the different values of the \textit{thresholds}. As the \textit{OpenYMSG} repository has only 195 revisions, we can apply the proposed technique to \textit{OpenYMSG} within a short period of time.

The preliminary experiment was conducted according to the following steps.

\textbf{STEP1:} The historical code repository of \textit{OpenYMSG} is divided into 5 equal parts based on the number of revisions, which we term: \( C_1, C_2, C_3, C_4 \) and \( C_5 \).

\textbf{STEP2 (the first-stage prediction):} The proposed technique builds first-stage prediction models. The models

\[ y_i = \beta + \sum_{j=0}^{29} \beta_j x_j \]
use changes in methods that occurred in $C_1, \ldots, C_{n-1}$ ($2 \leq n \leq 5$) as the training sets and predict whether changes in all methods in $C_n$ will be small or not. We set the threshold from 1 to 8, and we evaluate the accuracies of the prediction results yielded from the k-NN algorithm using three $k$ values ($k = 1, 3, 5$).

**STEP3:** The small changes in repository were extracted from each $C_n$ ($1 \leq n \leq 4$). The extracted changes are used as training sets to build second-stage prediction models.

**STEP4 (the second-stage prediction):** The proposed technique builds second-stage prediction models based on two methods: one method uses only the small changes; the other method uses all changes. These changes occurred in $C_1, \ldots, C_{n-1}$ ($2 \leq n \leq 5$) are used for building models and the models predict the next changes in methods that occurred in $C_n$.

**STEP5:** The predicted changes in $C_n$ are compared with the actual changes in $C_n$. This step yields three values. The first is the rate to which the actual small changes are correctly predicted as small (the accuracy of the first-stage prediction). The second is the rate to which changes are predicted correctly (the accuracy of the second-stage prediction). The third is the number of changes predicted correctly. The second and the third values regard a change as correct when the number of each program elements in the change is predicted perfectly. As $n$ is not less than 2 and not more than 5, we obtain four results from OpenYMSG.

In STEP2, we investigated the results yielded by the k-NN algorithm using the three $k$ values, and they were found to be almost the same. Therefore, we use $k = 1$ in these experiments because the k-NN algorithm using $k = 1$ can produce results in the shortest time. The output of the k-NN algorithm is categorized into four categories, as shown in Table 2. Then, precision is defined as follows:

$$
\text{Precision} = \frac{TP}{TP + FP}
$$

We considered precision to be important because the outputs are required to contain the correct values. If the outputs contain incorrect values, prediction models based on these incorrect values would lead to a wrong evolution. In order to achieve automated software evolution, incorrect evolutions must be avoided.

Suppose method $m$ is predicted to become $m'$ and actually changes to $m_{change}$ in the next change. Accuracy is defined as follows:

$$
\text{Accuracy} = \frac{\text{same}(v_{m'}, v_{m_{change}})}{|M_{all}|} (m \in M_{all})
$$

where, $M_{all}$ represents all the methods that have small changes occurred in each of $C_1 (1 \leq n \leq 5)$, and same($v_{m'}, v_{m_{change}}$) represents the method whose predicted number of program elements is equal to the actual one. We evaluate the results using precision and accuracy in the rest of this paper.

Figure 5 shows the result of the preliminary experiment. In each of the graphs, the horizontal axis represent the thresholds. The left vertical axis represent the number of small changes that occurred from the threshold specified by the horizontal axis. The right vertical axis represent precisions of the first-stage prediction and accuracies of the second-stage prediction. Figure 5 shows that the number of small changes increases and the accuracy of the second-stage prediction decreases as the threshold is increased. Moreover, precision remains almost unchanged even when threshold changes.

From Fig. 5, it is clear that small thresholds produce high precision and accuracy. However, the number of small changes decreases with the decrease in threshold. In other words, the number of methods that are targets for prediction decreases. Thus, we should investigate the relationship of precision/accuracy and the rate of small changes. Figure 6 shows two values. One is the product of precision of the first-stage and accuracy of the second-stage. From Fig. 6 it can be seen that the proposed technique can make correct predictions about 80% of the time when using 1 as threshold. The second value is the rate of small changes in all changes, and it is clear that the rate increases with the increase in threshold. From Fig. 5 and Fig. 6 it can be seen that the proposed technique has high precision and accuracy when the threshold is 1. However, at the same time, only about 20% PE-Vectors are targeted for prediction.

We think that 1 is rather strict as the threshold value. Suppose a case that a statement is inserted/deleted in a method, then the change will affect at least 5 kinds of program elements. However, the prediction model ignores the change if the threshold is 1. Hence, we have decided to use 5

---

**Table 1: Target Software Systems**

<table>
<thead>
<tr>
<th>Name</th>
<th>Start revision (date)</th>
<th>End revision (date)</th>
<th># of target revisions</th>
<th>LOC of end revision</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenYMSG</td>
<td>1 (2007/04/17)</td>
<td>195 (2010/12/07)</td>
<td>155</td>
<td>163,545</td>
</tr>
<tr>
<td>ArgoUML</td>
<td>1 (1998/01/27)</td>
<td>18,993 (2012/07/10)</td>
<td>3,918</td>
<td>355,411</td>
</tr>
<tr>
<td>Ant</td>
<td>267,549 (2000/01/13)</td>
<td>1,233,420 (2012/01/20)</td>
<td>8,284</td>
<td>255,169</td>
</tr>
</tbody>
</table>

---

**Table 2: Relationship between Predicted and Actual Values**

<table>
<thead>
<tr>
<th>predicted value</th>
<th>small</th>
<th>actual value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TP</td>
<td>FN</td>
</tr>
<tr>
<td>small</td>
<td>TP</td>
<td>FN</td>
</tr>
<tr>
<td>large</td>
<td>FN</td>
<td>TN</td>
</tr>
<tr>
<td>large</td>
<td>FN</td>
<td>TN</td>
</tr>
</tbody>
</table>

---

**Figure 6: Precision, Accuracy and Rate of small changes.**
as the threshold value as well as 1, even though the accuracy of the proposed technique may decrease.

Figure 7 shows the number of predicted PE-Vectors based on the models using all the changes in C5. The horizontal axis represents the number of program elements whose predicted values do not equal the actual values. The vertical axis represents the number of such PE-Vectors. Hence, the bar chart above 0 on the horizontal axis represents the number of PE-Vectors that are perfectly predicted. As compared to Fig. 5 and Fig. 6, it is difficult to predict large changes correctly and it is important to target only small changes.

The preliminary experiment is summarized as follows. In order to predict the number of all elements in PE-Vectors, it is important to use small changes as targets for prediction. In the main experiment, we use 1 and 5 as thresholds.

7. MAIN EXPERIMENT

In this experiment, we applied the proposed technique to ArgoUML and Ant using the thresholds decided by the preliminary experiment. The steps of the experiment are almost the same as for the preliminary experiment. The differences between them are the target software, the threshold, and conducting both predictions: that using all the elements, or not with an element. As a result, the proposed technique can predict the next change correctly when using all the elements about 85%-96% of the time.

Figure 9 shows the results of the second-stage prediction in Ant when threshold = 1. We consider that the results (93%-99%) are very good. The proposed technique can yield a good result because of two factors. One is that we set a strict threshold, and the other is that it is difficult to predict the number of low-level elements (e.g. SIMPLE_NAME or PRIMITIVE_TYPE) correctly because the elements appear in source code frequently. In particular, SIMPLE_NAME is used as a variable name, method name and so on. This fact makes prediction of the number of SIMPLE_NAME difficult.

Figure 11 shows the statement level results of the second-stage prediction in Ant when threshold = 5. As compared with the case of threshold = 1, the percentages are low. However, the proposed technique can target more PE-Vectors than in the case of threshold = 1. The experiment is summarized as follows. The proposed technique has 88%-97% precisions in the first-stage and 54%-96% accuracies in the second-stage when using all the elements. The accuracies change significantly according to the threshold. In comparison, the proposed technique has 92%-100% precisions in the first-stage and 73%-99% accuracies in the second-stage when using only the state-
The proposed method occasionally cannot detect the next change in a large method correctly. A large method has many SIMPLE_NAME. Predicting correctly the number of many SIMPLE_NAME is difficult because the elements appear in source code frequently.

8.2 Case of Failure

Figure 13 shows a prediction failure case. In this case, the number of SIMPLE_NAME changes from 53 to 57. The four additions SIMPLE_NAME are Thread, sleep and two U. However, such change did not occur in the training set. Therefore, the model predicted the number of SIMPLE_NAME not as 57 but as 56.

The proposed method occasionally cannot detect the next change in a large method correctly. A large method has many SIMPLE_NAME. Predicting correctly the number of many SIMPLE_NAME is difficult because the elements appear in

8. DISCUSSION

In this section, we discuss the cases of two prediction results. One is of a case of success, and the other is a case of failure.

8.1 Case of Success

Figure 12 shows a prediction success case. In this case,
PLE_NAME is the primary cause of prediction failure. To increase accuracy of prediction models, we need to tackle the problem of SIMPLE_NAME. Currently, we consider one solution, that is we use the AST treating variable name or method name as separate nodes. AST as provided by JDT treats the variable name or method name as same nodes, SIMPLE_NAME. If we use the enhanced AST, the number of SIMPLE_NAME in the next change could be predicted correctly.

9. THREATS TO VALIDITY
In this section, we describe two threats to validities.

9.1 Dividing Repositories
In this experiment, the historical code repositories were divided into 5 equal parts based on the number of revisions. However, if the repositories are divided according to different methods (e.g. based on version upgrade date or development periods) or if the number of divided repositories is different from this experiment, we might obtain different results. However, it is almost impossible to conduct experiments using all the combination of all possible methods of dividing and the possible number of divisions.

9.2 Building Models
In this experiment, only the small changes (threshold is 1 or 5) were used for the training set used to build prediction models. If the threshold is changed, different results are obtained. If the threshold is decreased, a stricter prediction is conducted. However, the number of methods that are targets for prediction would be decreased. Conversely, if the threshold is increased, the number of methods that are targets for prediction are increased; however, the accuracy of the prediction models would be decreased as large changes are included in the training set.

10. CONCLUSIONS
In this paper, we proposed a technique to predict the next change in source code. While existing techniques concentrate on changes for fixing bugs, our technique handles all kinds of changes. As an evaluation of the proposed technique, we conducted an experiment on three open source software systems. We performed two kinds of predictions in this experiment: one is predicting all kinds of program elements that are deleted and added in the next change; the other is predicting only program elements that are statements or large elements. The precision of the two predictions were 54%-96% and 73%-99%, respectively.

As the next step in this research, we are planning to predict (generate) the source code of the next version. There are other studies that generate source code (patches) for fixing bugs. They leverage genetic programming techniques. In our research, we also leverage genetic programming for the prediction.

11. ACKNOWLEDGMENTS
This work was supported by JSPS KAKENHI Grant Numbers 25220003, 24650011, and 24680002.

12. REFERENCES
**Figure 10:** Statement Level Prediction Results for Ant in the Second-Stage when Threshold = 1.

**Figure 11:** Statement Level Prediction Results for Ant in the Second-Stage when Threshold = 5.


[22] P. Tonella. Using a Concept Lattice of Decomposition Slices for Program Understanding and Impact


