Predicting Next Changes at the Fine-Grained Level

Hiroaki Murakami, Keisuke Hotta, Yoshiki Higo, Shinji Kusumoto
Graduate School of Information Science and Technology, Osaka University, 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. Such mistakes entail additional cost in having to reedit the source code correctly, and repeated changes themselves can be a vulnerability 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, we propose a technique to predict the types of program elements deleted and added in a next change to Java methods. This technique is designed to support developers in deciding how to change source code after they have identified a method to be changed. We evaluated predictions using the proposed technique with two thresholds, which are sizes of source code changes. For predictions with the smaller threshold where only a single type of program element was added or deleted, the accuracy of the proposed technique was 74%–85%. However, for the larger threshold, where 5 or fewer types of program elements were added or deleted, the accuracy was 44%–48%.**

***Keywords*—Fine-Grained Change Prediction, Automated Code Evolution, Static Code Analysis**

I. Introduction

Changing source code is the most costly activity in software maintenance [1], 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 they can degrade the software quality [2]. 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 some techniques that construct prediction models to predict those modules having frequent changes [3], [4], [5]. Another technique is tailored to interfaces, which are generally stable components in software systems [6]. Tsantalis et al. proposed a technique to calculate the probability that each class will be changed in a future version [7]. 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 reviewing them.

A hot research topic is identifying pairs or sets of modules that need to be changed together [8], [9], [10], [11]. After identifying the modules that need to be changed, developers can use the available techniques to know which modules need to be changed together. Such a support mechanism prevents developers from forgetting to change code fragments in a given task. Facilitating bug fixes is another particularly hot research topic. Various techniques are available, such as prioritizing open bugs for fixing [12], identifying code fragments that cause specific bugs [13], and validating whether bug fixes have been performed correctly [14].

Many approaches fix bugs automatically by generating patches for bug fixes. Perkins et al. developed ClearView, which analyzes a target program dynamically and generates patches that satisfy invariants in the program [15]. Wei et al. proposed a technique that generates patches that satisfy not only invariants but also preconditions and postconditions [16]. Because Wei's technique uses preconditions and postconditions, it is able to fix more bugs than can Perkin's technique. However, to be analyzed by Wei's technique, programs must have descriptions of invariants, preconditions, and postconditions in their source files. In Perkin's technique, invariants are automatically identified by using the tool Daikon [17]; 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 [18]. Weimer et al. developed a tool called GenProg that uses genetic programming to generate patches for fixing bugs [19], [20] and reported that GenProg was able to fix 55 out of 105 bugs automatically [21]. 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 the 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*" [22].

In this research, we first investigated 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 developed 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 it works particularly well for 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) program elements that will be added or deleted in the next version. If the predicted source code satisfies his/her requirements, he/she can obtain a patch to change the current source code in 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 predicts the next changes. Currently, our technique predicts the types of program elements that will be added and deleted in the next change.
- We evaluate our technique by conducting an experiment on two open-source software systems. In the experiment, predictions are performed with two thresholds, which are

the size of the source code changes. The smaller threshold is intended to evaluate predictions where source code changes are very small. That is, only a single type of program elements is added or deleted. The larger threshold is intended to evaluate predictions where one program statement is added, deleted, or changed. The accuracies of the two predictions are 74%–85% and 44%–48%, respectively.

The remainder of this paper is organized as follows: Section II introduces related work and discusses the differences between them and our technique. Section III provides an overview of our technique, and Section IV describes how to build prediction models. Section V shows how to implement our technique. Section VI reports the experiment and the experimental result in detail. Section VII discusses the experimental result by using examples of source code. We present our final goal and paths to the goal in Section VIII. Section IX describes the threats to validity. Section X concludes the paper and states future work.

II. Related Work

A number of approaches use source code metrics to train prediction models that can lead developers to the change-prone and fault-prone modules in a software system [3], [4], [5], [6]. Although the experimental results of these approaches are promising, they do not provide insights into the details of the changes.

Giger et al. explored prediction models for determining 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 [23]. Their models output a list of potentially change-prone files ranked according to their change-proneness overall and their change type category. The purpose of their research and ours is the same, which is to provide insights into 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 the types of program elements that are deleted and added.

Goues et al. proposed a technique to repair programs automatically [19], [20]. 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 a repaired version of the source code itself. 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 [24]. 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.

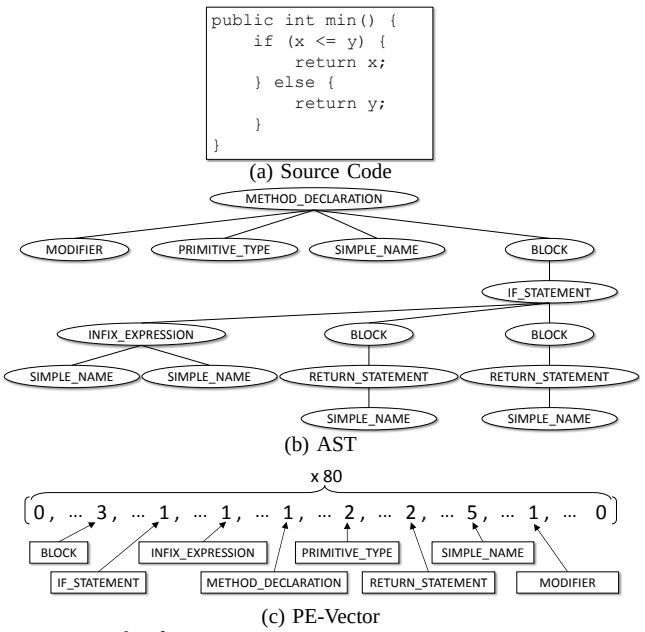


Fig. 1. Example of a PE-Vector

Gethers et al. proposed a technique to enhance impact analysis by combining information retrieval, dynamic analysis and mining software repositories techniques [25], [26]. Their technique identified a set of methods that should be fixed in order to achieve the given change request (e.g., a bug report). They showed that their technique provided an improved accuracy over the previously approaches.

III. Overview of Our Technique

This paper proposes a technique to predict how to change a given Java method at the program element level. As an example, 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 input a Java method that needs a change. The proposed technique then predicts the next change on the given method with the prediction models. This section introduces terms that are used in this paper, followed by an overview of the procedure of the proposed technique.

A. 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 and defines 83 types of AST nodes, including three related to comments[1]. We use 80 types of program elements because we are not interested in comments. This paper uses symbols $A_0, \cdots, A_{79}$ to represent the different types of program element.

[1]The types of AST nodes in JDT and this study are defined in the class `org.eclipse.jdt.core.dom.ASTNode`. A list of nodes is provided in the Eclipse 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.

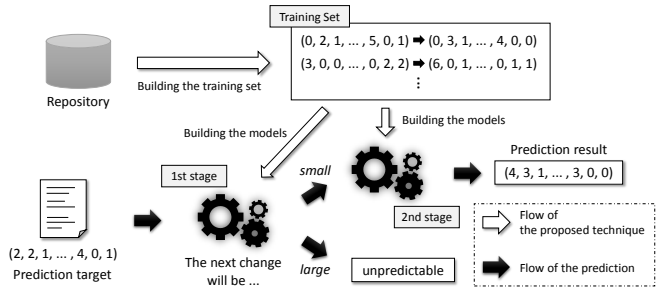


Fig. 2. Overview of Proposed Technique

PE-Vector: The proposed technique treats each method as a vector named a Program Element Vector (**PE-Vector**). A PE-Vector is a numerical vector that has 80 dimensions, and each element of a PE-Vector represents the number of each program element in the method. This paper describes the PE-Vector of a method m as $\vec{v_m} = (x_0, \cdots, x_{79})$. Here, x_i in $\vec{v_m}$ denotes the number of the 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, due to the space limitation, we omitted the attributes from the figures whose value is 0, except for the head and the tail of the vector.

B. Overview of the Procedure

Figure 2 provides an overview of the procedure of the proposed technique and a prediction made by using it. As the first step in the 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. Each element of the training set is a pair of PE-Vectors v_m and $v_{m'}$, where m was changed to m'. Note that the PE-Vectors before and after a change might be the same vector. This means that the change did not alter the numbers of any program elements. For example, this case occurs when the change is the renaming of local variables. With the training set, the proposed technique builds the prediction models. The models tell us what kind of change will occur next in a given method. The proposed technique adopts a two-stage prediction, as described below.

Prediction for Filtering: the proposed technique predicts 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: the proposed technique predicts the quantity of the next change in terms of AST nodes for a given method. That is, the proposed technique predicts the next PE-Vector of a given method m. For example, if we suppose that m becomes m' in the next change, then the proposed technique predicts $\vec{v_{m'}}$ based on $\vec{v_m}$. This stage targets only methods that are predicted as undergoing *small* changes in the first-stage 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 for automated source code changes. The proposed technique is the first step of automated source code changes. Hence, the accuracy of a prediction made with the proposed technique is important for the final goal. That is, if an incorrect prediction is made, this would lead to an incorrect change. We believe that an incorrect change must be avoided even though we will lose some candidates that might be able to be changed automatically. Therefore, we decided to ignore *large* changes currently by such filtering. However, we will improve our technique to be able to predict *large* changes correctly as a future task. The above leads to 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 is changed and becomes m', and $\vec{v} = (x_0, \ldots, x_{79})$ is a difference vector from m to m' ($\vec{v} = \vec{v'_m} - \vec{v_m}$). The change to m is regarded as *small* if the following condition holds; otherwise, it is regarded as *large*.

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

where

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

We use the number of types of program elements instead of code churn, which is a common metric for measuring the size of a change. 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. Specifically, the proposed technique can predict that "an `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 by a feature we can predict. In this case, we 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 the renaming of variables. This result does not indicate that the method does not need any changes. This is because the proposed technique assumes that its users will input methods that should be changed. Predicting whether a given method needs any changes is not within the scope of the proposed technique.

C. Restrictions

The proposed technique 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 the type of program elements that are added/deleted in the next change.

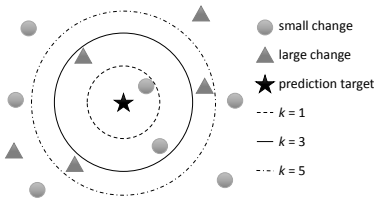


Fig. 3. The k-NN Algorithm

IV. Prediction Models

This section describes how to build prediction models by 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 if they will undergo *large* changes. The second stage predicts a PE-Vector of the method m' that a given method m will become by the next change. The following subsections describe each stage in detail.

A. Prediction for Filtering

The first-stage prediction uses the k-nearest neighbor (k-NN) algorithm to predict if the next change in a method will be *small* or *large*. This is a nonparametric algorithm for classification and regression, and it is the simplest of the machine learning algorithms. We use this algorithm because of its high scalability. The k-NN algorithm makes a prediction for a given object based on the majority vote by its k closest objects in the training data in vector space. In our case, the objects are PE-Vectors, and the distance between two objects is measured by the Euclidean distance between the two PE-Vectors.

Figure 3 shows the k-NN algorithm. In the case of $k = 1$, the algorithm chooses the nearest object to a given object from the training set. In the example in Fig. 3, the nearest object is marked as *small*, and so the prediction result becomes *small*. In the case of $k = 3$, the prediction result is decided based on the majority vote by the 3 nearest objects. The prediction result becomes *small* because there are 2 *small* changes and 1 *large* change in the solid circle. The prediction in the case of $k = 5$ is made in a similar way. In this case, the algorithm regards the prediction target as *large* because the number of *small* changes is less than that of *large* changes in the outmost circle.

B. Prediction of Changes

The second-stage 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 the PE-Vector. In other words, it makes predictions on each program element independently. Hence, this stage generates the same number of regression functions as the dimension number of the PE-Vector, which is 80.

Let $\vec{v_m} = (x_0, \cdots, x_{79})$ be the target of prediction, and $\vec{v'_m} = (y_0, \cdots, y_{79})$ be a PE-Vector created by the prediction for $\vec{v_m}$. The proposed technique builds a regression function for each of the attributes $A_0, \ldots, A_{79}$. Equation (3) is the form of each regression function.

$$y_i = \beta + \sum_{j=0}^{79} \beta_j \, x_j \tag{3}$$

Equation (3) states that the value of an attribute A_i is estimated with those of all the attributes $A_0, \ldots, A_{79}$. However, we should not use all the attributes as explanatory variables because some of the attributes are correlated. Hence, we have to select the variables to be used as explanatory variables to avoid multicollinearity. Therefore, β_j will have a nonzero value if the attribute A_j is selected as an explanatory variable; otherwise, β_j is 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 regression function for y_k.

V. 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 a static analysis to build prediction models to the `R` functions.

To build training sets, 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 detection [27]. The clone tracking technique was originally used to map code clones between two consecutive revisions, but it can be used to map methods between two consecutive revisions. It is based on CRD [28], 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 the code fragments. The technique can map clones well, even in the case where files having clones were moved or renamed because it uses the similarity of CRDs, not their perfect matching. As in the clone mapping technique, the proposed technique in this study calculates CRDs from every method and makes links of the methods between two consecutive revisions based on the similarity of their CRDs.

We used the `knn` function of `R` to predict *small* changes, and the `lm` function to perform the linear regression analysis. In addition, we adopted the method of increasing and decreasing the variables of the `step` function to select explanatory variables. The `step` function makes prediction models that have the lowest Akaike information criterion (AIC), which helps to avoid the problems of overfitting and multicollinearity.

VI. Experiment

The purpose of the experiment is to investigate to what extent the proposed technique can correctly predict the number of types of program elements in the next change. In this experiment, we use *ArgoUML* and *Ant* as the target software systems. These systems are written in Java and managed by `Subversion`. Table I shows the details of the systems. This section consists of the following two experiments.

[2]http://www.r-project.org/

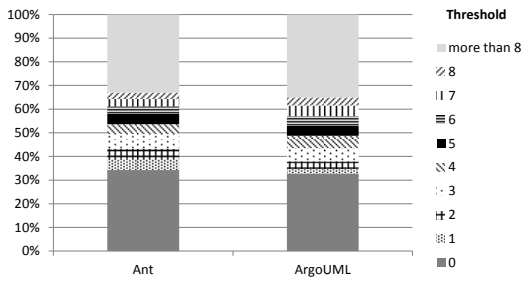


Fig. 4. Rates of *Small* Changes

Investigation of small changes: first, we investigated how many *small* changes account for all the changes in the target software systems. If few *small* changes are used for building the prediction models, the proposed technique does not work well. Therefore, we report the rates of *small* changes against all changes and introduce some examples of *small* changes.

Prediction of program elements: the proposed technique predicts whether the next changes are *small* for the target software systems. If the next changes are predicted as *small*, the proposed technique predicts the types and the number of program elements in the next changes.

A. Investigation of small changes

We calculated the rates of *small* changes against all changes by using various thresholds in Equation (1). Figure 4 shows the rates. As this figure shows, in the case of *Ant*, the percentage of *small* changes is approximately 35% when the threshold is 0 and approximately 39% when the threshold is 1. When the threshold is 0, the number of AST nodes does not change. Such changes include modifying variable names and reordering statements. From Fig. 4, the percentages of *small* changes exceed 50% when the threshold is 5. It is interesting that the number of *small* changes accounts for over half of all changes when the threshold is 5. We think 1 is rather strict as a threshold value. Suppose a statement is changed in a method, then the change may affect about 5 types of program elements. However, the prediction model ignores the change if the threshold is 1. Hence, we decided to use 1 and 5 as threshold values.

Figure 5 shows two examples of code changes when the number of types of changed AST nodes is 1. The examples were obtained from *ArgoUML*. In the case of Fig. 5(a), a `null` argument is deleted from a method invocation. In terms of the AST, the change decreases the number of `null` literal nodes. In Fig. 5(b), braces are added to all `if` statements, which increases the number of block nodes in AST. Figure 6 shows two examples of code changes when the number of types of changed AST nodes is 5. Figure 6(a) shows a change that adds an `if` statement. This change affects the number of AST nodes of the `continue` statement, `if` statement, method invocation, number literal, and method name. Figure 6(b) shows another change that modifies a `return` statement by uncommenting. It increases the number of five types of AST nodes: infix expression, `instanceof` operator, parenthesized expression, variable name, and type name.

TABLE I. TARGET SOFTWARE SYSTEMS

Name	Start revision (date)	End revision (date)	# revisions
Ant	267,549 (2000/01/13)	1,233,420 (2012/01/20)	8,284
ArgoUML	1 (1998/01/27)	19,893 (2012/07/10)	3,918

```
 public void actionPerformed(ActionEvent e) {
    System.out.println("making class...");
-   _cmdCreateNode.doIt(null);
+   _cmdCreateNode.doIt();
  }
```

(a) Deleting a `null` Argument from a Method Invocation

```
 public void setOwner(Object node) {
   Object oldOwner = getOwner();
-  if (oldOwner != null && oldOwner instanceof GraphNodeHooks)
+  if (oldOwner != null && oldOwner instanceof GraphNodeHooks) {
     ((GraphNodeHooks)oldOwner).removePropertyChangeListener(this);
+  }
-  if (oldOwner != null && oldOwner instanceof Highlightable)
+  if (oldOwner != null && oldOwner instanceof Highlightable) {
     ((Highlightable)oldOwner).removePropertyChangeListener(this);
+  }
-  if (node instanceof GraphNodeHooks)
+  if (node instanceof GraphNodeHooks) {
     ((GraphNodeHooks)node).addPropertyChangeListener(this);
+  }
-  else if (node instanceof Highlightable)
+  else if (node instanceof Highlightable) {
     ((Highlightable)node).addPropertyChangeListener(this);
+  }
   super.setOwner(node);
 }
```

(b) Adding Braces to All `if` Statements

Fig. 5. Examples of *Small* Changes when Threshold is 1

```
 public boolean predicate2(Object dm, Designer dsgr) {
   if (!(dm instanceof Classifier)) return NO_PROBLEM;
      ⋮
     if (Name.UNSPEC.equals(aeName)) continue;
       String aeNameStr = aeName.getBody();
+      if (aeNameStr.length() == 0) continue;
       if (namesSeen.contains(aeNameStr)) return PROBLEM_FOUND;
       namesSeen.addElement(aeNameStr);
     }
   }
   return NO_PROBLEM;
 }
```

(a) Adding an `if` Statement

```
 public boolean shouldBeEnabled() {
   ProjectBrowser pb = ProjectBrowser.TheInstance;
   Project p = pb.getProject();
   Object target = pb.getDetailsTarget();
-  return super.shouldBeEnabled() && p != null;
-  // && (target instanceof ModelElement);
+  return super.shouldBeEnabled() && p != null &&
+         (target instanceof ModelElement);
  }
```

(b) Modifying a `return` Statement by Uncommenting

Fig. 6. Examples of *Small* Changes when Threshold is 5

B. Prediction of program elements

In this subsection, we report two types of prediction results. The first is a prediction of whether the next changes are *small*. The second is a prediction of the types and the number of program elements in the next change. Based on the prediction results, we answer the following two research questions (RQs).

RQ1: how accurately does the proposed technique predict whether the next change will be *small*? (Accuracy of the first-stage prediction)

RQ2: how accurately does the proposed technique predict the types and the number of program elements in the next changes? (Accuracy of the second-stage prediction)

In preparation for the experiment, the historical code repositories of each target software system were divided into 5 equal parts based on the number of revisions, which we named C_1, C_2, C_3, C_4, and C_5. In the rest of this subsection, we describe the steps of the experiment and report the prediction results.

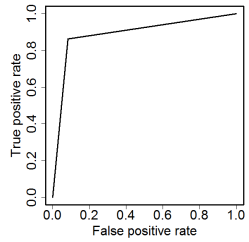


Fig. 7. The ROC Curve Obtained from *Ant*

1) Prediction whether the next changes are small: the proposed technique built first-stage prediction models for predicting whether the next changes are *small*. The models were built by the training sets $C_1, \ldots, C_{n-1} (2 \leq n \leq 5)$. Then, the models predicted whether the changes in C_n would be *small*. We evaluated the accuracies of the prediction results yielded from the k-NN algorithm by using three *k* values (*k* = 1, 3, 5). However, the results were almost the same. Therefore, we use *k* = 1 in this experiment because the k-NN algorithm using *k* = 1 can produce results in the shortest time. The output of the k-NN algorithm was classified into four categories, as shown in Table II. We obtained a ROC (Receiver Operating Characteristic) curves from the classification results. The ROC curve is created by plotting the fraction of true positives out of the total actual positives (true positive rate) and the fraction of false positives out of the total actual negatives (false positive rate). Figure 7 shows the ROC curve obtained from *Ant* in C_2. Moreover, we obtained an AUC (Area Under the Curve) from the ROC curve. AUC is a commonly used evaluation scale for binary prediction models.

2) Prediction of the types and the number of program elements in the next change: the proposed technique built second-stage prediction models for predicting the types and the number of program elements in the next change. In this experiment, a predicted change is regarded as correct when the number of each program element in the predicted change is equal to the actual change. This prediction was conducted according to the following steps.

STEP1: *small* changes in the target software systems were extracted from each $C_n (1 \leq n \leq 4)$.

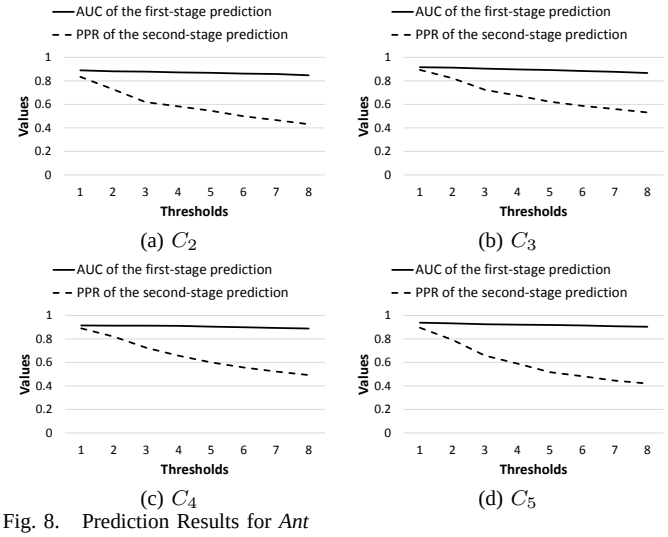
STEP2: the proposed technique built second-stage prediction models by using the *small* changes extracted in STEP1. These changes occurred in $C_1, \ldots, C_{n-1} (2 \leq n \leq 5)$ and were used for building models, and the models predicted the next changes that occurred in C_n.

STEP3: the predicted changes in C_n were compared with the actual changes in C_n. This step yielded two values. The first was the rate at which the changes were predicted correctly. The second was the number of changes predicted correctly. As n was either 2, 3, 4, or 5, we obtained four results from each target software system.

In this evaluation, we defined PPR (Perfectly Predicted Ratio). PPR is the ratio of perfectly predicted PE-Vectors to all the PE-Vectors in each C_n. The perfectly predicted PE-Vectors represent PE-Vectors whose predicted number of types of program elements is equal to the actual ones in the next change. We consider that PPRs are the performances of the second-stage prediction models.

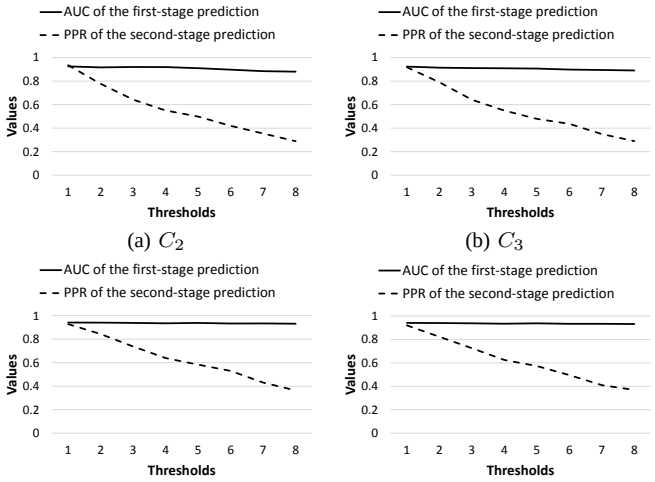
TABLE II. RELATIONSHIP AMONG PREDICTED AND ACTUAL VALUES

		Actual	
		small	*large*
Predicted	*small*	True Positives (TP)	False Positives (FP)
	large	False Negatives (FN)	True Negatives (TN)



(a) C_2 (b) C_3 (c) C_4 (d) C_5

Fig. 8. Prediction Results for *Ant*



(a) C_2 (b) C_3 (c) C_4 (d) C_5

Fig. 9. Prediction Results for *ArgoUML*

3) Experimental Results: Figures 8 and 9 show AUCs and PPRs for the target software systems. In each graph, the solid lines represent AUCs and the dashed lines represent PPRs. From these graphs, we found that AUCs change little as the threshold increases and PPRs decrease as the threshold increases. Specifically, when the threshold is 1, AUC varies from 89% to 94% for *Ant* and from 92% to 94% for *ArgoUML*. PPR varies from 83% to 90% for *Ant* and from 92% to 94% for *ArgoUML*. When the threshold is 5, AUC results are 87%–92% for *Ant* and 91%–94% for *ArgoUML*. PPR results are 55%–62% for *Ant* and 48%–58% for *ArgoUML*. The product of AUC and PPR is regarded as the performance of the whole prediction model. Thus, the performance of the prediction model for *Ant* is 74% (= 89% $\times$ 83%), and that for *ArgoUML* is 85% (= 92% $\times$ 92%) when the threshold is 1. When the threshold is 5, these percentages are 48% for *Ant* and 44% for *ArgoUML*. Based on the experimental results, we answer the research questions as follows.

```
  public void testSendReceiveStartNotifyTyping() throws IOException {
    sess1.sendTypingNotification(OTHERUSR, true);
     ⋮
    assertTrue(sne.isTyping());
-   assertTrue(sne.getMode()==1);
+   assertTrue(sne.isOn());
    sess1.sendMessage(OTHERUSR, CHATMESSAGE,sess1.getLoginIdentity());
     ⋮
    assertEquals(event.getEvent().getMessage(), CHATMESSAGE);
  }
```

Fig. 10. Prediction Success

```
  public void testDuplicateLogins() throws Exception {
+   Thread.sleep(500);
-   final Session<RosterV1> sessionOne = createSession();
-   final Session<RosterV1> sessionTwo = createSession();
+   final Session<T, U> sessionOne = createSession();
+   final Session<T, U> sessionTwo = createSession();
    assertEquals(SessionState.UNSTARTED, sessionOne.getSessionStatus());
     ⋮
    assertEquals(SessionState.UNSTARTED, sessionTwo.getSessionStatus());
  }
```

Fig. 11. Prediction Failure

RQ1: the proposed technique predicts whether the next change will be *small* with 89%–94% as AUC for *Ant* and 92%–94% for *ArgoUML* when the threshold is 1. When the threshold is 5, these percentages are 87%–92% for *Ant* and 91%–94% for *ArgoUML*.

RQ2: the proposed technique predicts the number of program elements in the next change, with 83%–90% as PPR for *Ant* and 92%–94% for *ArgoUML* when the threshold is 1. When the threshold is 5, these percentages are 55%–62% for *Ant* and 48%–58% for *ArgoUML*.

VII. Discussion

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

A. Case of Success

Figure 10 shows the success case. In this case, the number of infix expressions changes from 1 to 0, and the number of number literals changes from 3 to 2. In the training set, a change in the number of infix expressions from 1 to 0 occurred twice, and that of the number of number literals from 3 to 2 occurred 62 times. However, these changes did not occur at the same time in the training set. Thus, existing techniques using a fix pattern (e.g., [24]) cannot find this case. The reason the proposed technique could find this case is that the prediction models were built by learning the changes of each of the program elements. This is the largest advantage of the proposed technique.

B. Case of Failure

Figure 11 shows the failure case. In this case, the number of `SIMPLE_NAME` nodes changes from 53 to 57. `SIMPLE_NAME` represents a user-defined name (e.g., variable name or method name). The added `SIMPLE_NAME` occurrences are "Thread", "sleep", and two "U". However, such changes did not occur in the training set. Therefore, the model predicted the number of `SIMPLE_NAME` as 56 and not as 57. The proposed technique occasionally cannot correctly detect the next change in a large method. A large method has many occurrences of `SIMPLE_NAME`. Predicting correctly the number of many `SIMPLE_NAME` occurrences is difficult because the change from a large number to another large number (e.g., from 53 to 57) seldom occurs and the proposed technique cannot build correct models. In the experiment, we found that `SIMPLE_NAME` is the primary cause of prediction failure. To increase the $accuracy$ of the prediction models, we need to tackle the problem of `SIMPLE_NAME`. Currently, we consider one solution, which is using the AST that treats the variable name or the method name as separate nodes. The AST provided by JDT treats the variable name or method name as the same node, `SIMPLE_NAME`. If we use the enhanced AST, the number of `SIMPLE_NAME` occurrences in the next change could be predicted correctly.

VIII. Our Goal

Our final goal is the realization of automated code changes. However, at this time, we only achieved predicting the number of added/deleted AST nodes in the next change. To achieve automated code changes, we have to tackle following problems, (1) where the change should be made and (2) what actual code is added, deleted and modified. In order to resolve these problems, we have to expand the proposed technique. For example, we will predict the locations of program elements in the next change by using a rich PE-Vector that contains locational information. Moreover, we will convert some elements of the PE-Vector into an actual code (e.g. if one `IF_STATEMENT`, one `SIMPLE_NAME` and one `BLOCK` are added, the following code would be created).

```
if (var) {}
```

Another solution is applying the idea of the proposed technique for the existing method. GenProg uses genetic algorithm to randomly generate bug patches. If it is found that what kinds of program elements are likely changed by the proposed technique, the genetic algorithm can focus more program elements than random generations.

IX. Threats to validity

In this section, we describe some threats to validities.

A. Target Software Systems

In this experiment, each of the target repositories was divided into 5 equal parts based on the number of revisions. However, if the repositories are divided in different ways (e.g., based on the version upgrade date or development periods) or if the number of divided repositories is different from that in this experiment, we might obtain different results. Moreover, we used only two software systems for the experiment. When other software systems are used, the proposed technique may yield different results from that reported in the experiment. In order to get rid of the threat, we have to evaluate the proposed technique with more than two software systems.

B. Building Models

In this experiment, only the *small* changes (threshold is 1 or 5) were used for the training set. If the threshold is changed, different results will be 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 decrease due to *large* changes included in the training set.

C. Large Changes

The proposed technique classified all changes into *small* changes or *large* changes. Then, we used only *small* changes for this experiment. In other words, we ignored *large* changes. The reason is that we consider that *small* changes make prediction models more reliable than do *large* changes. To achieve a perfect prediction, *large* changes should not be ignored. Prediction using *large* changes is our future work.

D. Types of Changes

The proposed technique did not consider a context or a requirement of the change. For example, if a change causes bugs, the proposed technique uses the change for building the prediction models because the proposed technique did not consider the context of the change. In that case, the proposed technique has a risk of predicting the change that causes bugs. Furthermore, the proposed technique did not consider a contributor. In the case of predicting the same or similar change as in the past, if the contributor of the past change and that of the present change are same, the prediction models are likely to have high accuracies. On the other hand, if the present contributor is different from the one of the past change, the prediction may result in failure. Therefore, it is difficult for the proposed technique to predict the change in the software systems that many people develop within a certain amount of time. One of the solution for the problem is that the proposed technique builds prediction models by using only the changes that the contributor of the changes and the user of the proposed technique are same.

X. 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 two open source software systems. We performed predictions by the proposed technique with two thresholds in this experiment: one was predicting only very small changes, where only a single type of program element was added or deleted in the next change; the other was intended to predict changes where 1 program statement was added, deleted, or changed. The accuracies of the predictions were 74%–85% and 44%–48%, respectively. As the next step in this research, we are planning to predict (generate) the source code of the next version.

Acknowledgment

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

References

[1] R. C. Seacord, D. Plakosh, and G. A. Lewis, *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[2] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy, "Change Bursts As Defect Predictors," in *ISSRE'10*, pp. 309–318.

[3] E. Arisholm, L. C. Briand, and A. Foyen, "Dynamic Coupling Measurement for Object-Oriented Software," *TSE*, vol. 30, no. 8, pp. 491–506, 2004.

[4] M. Dagpinar and J. H. Jahnke, "Predicting Maintainability with Object-Oriented Metrics - An Empirical Comparison," in *WCRE'03*, pp. 155–164.

[5] W. Li and S. Henry, "Object-oriented Metrics That Predict Maintainability," *Journal of Systems and Software*, vol. 23, no. 2, pp. 111–122, 1993.

[6] D. Romano and M. Pinzger, "Using Source Code Metrics to Predict Change-prone Java Interfaces," in *ICSM'11*, pp. 303–312.

[7] N. Tsantalis, A. Chatzigeorgiou, and G. Stephanides, "Predicting the Probability of Change in Object-Oriented Systems," *TSE*, vol. 31, no. 7, pp. 601–614, 2005.

[8] H. Kagdi, "Improving Change Prediction with Fine-grained Source Code Mining," in *ASE'07*, pp. 559–562.

[9] R. Robbes, D. Pollet, and M. Lanza, "Logical Coupling Based on Fine-Grained Change Information," in *WCRE'08*, pp. 42–46.

[10] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll, "Predicting Source Code Changes by Mining Change History," *TSE*, vol. 30, no. 9, pp. 574–586, 2004.

[11] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller, "Mining Version Histories to Guide Software Changes," in *ICSE'04*, pp. 563–572.

[12] J. Anvik, L. Hiew, and G. C. Murphy, "Who Should Fix This Bug?" in *ICSE'06*, pp. 361–370.

[13] D. Saha, M. G. Nanda, P. Dhoolia, V. K. Nandivada, V. Sinha, and S. Chandra, "Fault Localization for Data-centric Programs," in *FSE'11*, pp. 157–167.

[14] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram, "How Do Fixes Become Bugs?" in *FSE'11*, pp. 26–36.

[15] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard, "Automatically Patching Errors in Deployed Software," in *SOSP'09*, pp. 87–102.

[16] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller, "Automated Fixing of Programs with Contracts," in *ISSTA'10*, pp. 61–72.

[17] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon System for Dynamic Detection of Likely Invariants," *Science of Computer Programming*, vol. 69, no. 1-3, pp. 35–45, 2007.

[18] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit, "Automated Atomicity-violation Fixing," in *PLDI'11*, pp. 389–400.

[19] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A Generic Method for Automatic Software Repair," *TSE*, vol. 38, no. 1, pp. 54–72, 2012.

[20] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically Finding Patches Using Genetic Programming," in *ICSE'09*, pp. 364–374.

[21] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each," in *ICSE'12*, pp. 3–13.

[22] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the Naturalness of Software," in *ICSE'12*, pp. 837–847.

[23] E. Giger, M. Pinzger, and H. C. Gall, "Can We Predict Types of Code Changes? An Empirical Analysis," in *MSR'12*, pp. 217–226.

[24] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *ICSE'13*, pp. 802–811.

[25] M. Gethers, H. H. Kagdi, B. Dit, and D. Poshyvanyk, "An adaptive approach to impact analysis from change requests to source code," in *ASE'11*, pp. 540–543.

[26] M. Gethers, B. Dit, H. H. Kagdi, and D. Poshyvanyk, "Integrated impact analysis for managing software changes," in *ICSE'12*, pp. 430–440.

[27] Y. Higo, K. Hotta, and S. Kusumoto, "Enhancement of CRD-based Clone Tracking," in *IWPSE'13*, pp. 28–37.

[28] E. Duala-Ekoko and M. P. Robillard, "Clone Region Descriptors: Representing and Tracking Duplication in Source Code," *TOSEM*, vol. 20, no. 1, pp. 3:1–3:31, 2010.