PAPER

Slicing Fine-Grained Code Change History

Katsuhisa MARUYAMA[†a)], **Takayuki OMORI**[†], ***Members***, ***and*** **Shinpei HAYASHI**[††], ***Nonmember***

SUMMARY Change-aware development environments can automatically record fine-grained code changes on a program and allow programmers to replay the recorded changes in chronological order. However, since they do not always need to replay all the code changes to investigate how a particular entity of the program has been changed, they often eliminate several code changes of no interest by manually skipping them in replaying. This skipping action is an obstacle that makes many programmers hesitate when they use existing replaying tools. This paper proposes a slicing mechanism that automatically removes manually skipped code changes from the whole history of past code changes and extracts only those necessary to build a particular class member of a Java program. In this mechanism, fine-grained code changes are represented by edit operations recorded on the source code of a program and dependencies among edit operations are formalized. The paper also presents a running tool that slices the operation history and replays its resulting slices. With this tool, programmers can avoid replaying nonessential edit operations for the construction of class members that they want to understand. Experimental results show that the tool offered improvements over conventional replaying tools with respect to the reduction of the number of edit operations needed to be examined and over history filtering tools with respect to the accuracy of edit operations to be replayed.

key words: *software maintenance and evolution, program comprehension, integrated development environments, program slicing, code changes*

1. Introduction

In software development and evolution, programmers (developers and maintainers) often confront a situation where they must understand existing code that has been written by themselves or someone else [1]. LaToza and Myers found in programmers' work hard-to-answer questions about the history of code: when, how, by whom, and why this code was changed or inserted [2]. Among these questions, one about why a certain code was implemented in a specific way is an especially serious problem for programmers [3], [4]. Their study also found that programmers wanted to know the entire history of a piece of code rather than its latest change. In other words, it is worthwhile to support the exploration of code history to answer questions about code [5]. Suitable time-series historical information simplifies the understanding of the existing code. As a result, it will facilitate programmers' tasks of reusing or modifying existing code in the future.

To assist code history exploration, change-based support has recently become available [6], [7]. For example, SpyWare [8], Syde [9], Fluorite [10], CodingTracker [11], and OperationRecorder [12] are embedded into modern integrated development environments (IDEs) and capture all of the fine-grained code changes performed on the editors provided by their respective IDEs. In addition, some of these recording tools collaborate with tools that visualize, filter, and/or replay recorded code changes [13]–[16].

Using these tools helps programmers keep track of the fine-grained code changes individually stored in a code's history and look at their chronological sequence. For example, a controlled experiment conducted by Hattori et al. demonstrated that the chronological replaying of fine-grained code changes outperforms existing commit-based versioning systems such as CVS [17] or Subversion [18] by helping programmers find answers to questions related to software evolution [19]. In addition, Parnin and DeLine investigated what is needed to assist programmers when they resume interrupted programming tasks [20]. As a result, they strongly prefer two different cues. One shows a chronologically sorted list of the programmer's activities, such as code selections, code edits, and saves. This emphasizes the helpfulness of chronologically replaying the code changes.

Although chronologically replaying fine-grained code changes is useful for understanding how the code was written and modified, we focus on the possibility of improving the assistance for such a replay. In general, replaying is time-consuming. If a huge amount of code changes were recorded, it would take a long time to replay every one. In most cases, programmers do not need to investigate the whole history of the code. They incrementally obtain knowledge on past code changes by partially replaying them, depending on their interests. To encourage programmers to exploit current replaying tools, the automatic extraction of code changes to be replayed is required [21]. This helps them efficiently explore the history of a particular block of code and understand its evolution. Consequently, this idea would alleviate the effort needed for the time-consuming task of replaying.

This paper proposes a mechanism that automatically extracts a collection of fine-grained code changes, all of which may be related to a particular program entity from the recorded change history. This mechanism is inspired by the concept of program slicing [22], which extracts from the code of a program a set of statements that might af-

Manuscript received July 17, 2015.
Manuscript revised November 13, 2015.
Manuscript publicized December 21, 2015.

[†]The authors are with Dept. Computer Science, Ritsumeikan University, Kusatsu-shi, 525–8577 Japan.
[††]The author is with Dept. Computer Science, Tokyo Institute of Technology, Tokyo, 152–8552 Japan.
a) E-mail: maru@cs.ritsumei.ac.jp

DOI: 10.1587/transinf.2015EDP7282

fect (the calculation of) the value of a variable of interest at a specified program point. The extracted code, which is called a program slice, is computed by a graph reachability algorithm for the program dependence graph of a target program. Here, the principal concern is simplification by program slicing to increase program comprehension [23], although various kinds of applications of program slicing have been proposed. Our idea exploits this simplification power of such slicing in replaying past code changes.

In our proposed mechanism, fine-grained code changes are represented by edit operations on the source code of a program. Moreover, a special graph, called an *operation history graph* (OpG), is introduced. It links class members (methods and fields) of the snapshots of the source code of a Java program by edit operations performed on their respective snapshots. Each vertex of an OpG represents either a class member or an edit operation. Edges denote the relationships among class members and edit operations. By traversing vertices and edges of an OpG, the mechanism only extracts the edit operations that are necessary to build (create, remove, and modify) a class member of interest from the history that consists of all the recorded edit operations. Such an extraction process and a collection of the extracted edit operations are called *operation history slicing* and an *operation history slice*, respectively. This paper also presents a running tool, OperationSliceReplayer, that slices the operation history and replays its resulting slices.

One important feature of our proposed slicing is that the contents restored by replaying only the edit operations included in an operation slice for a target class member on an arbitrary snapshot of source code are usually identical to the original contents of the target class member. This feature is derived from a property where the value resulting from the execution of a program slice for a target variable is always the same as the value of the target variable resulting from the execution of the original program. Due to this feature, if a programmer wants to understand a particular class member in her task, only the edit operations included in its operation history slice are replayed. In other words, she can avoid replaying any nonessential edit operations of class members that she has no interest in. Consequently, OperationSliceReplayer makes her program understanding task more efficient.

The basic mechanism of *operation history slicing* and its naive prototype were previously proposed [21]. Yet the previous paper lacked the detailed algorithms adopted by the mechanism and evaluation results. The main contributions of this paper include:

- A detailed explanation of the concept of *operation history slicing* and its mechanism.
- A sophisticated running tool that implements the proposed mechanism.
- Evaluation results that show the effectiveness of the tool.

The remainder of our paper is organized as follows. Section 2 introduces the conventional tools that can replay fine-grained code changes and describes a motivating example that shows the inconvenience of using them. Section 3 defines an operation history graph and operation history slicing based on this graph. Section 4 explains the implementation of a tool that supports the efficient replay of edit operations. Section 5 assesses the tool's effectiveness by showing experimental results. Section 6 presents related work. Finally, Sect. 7 concludes with a brief summary and immediate future work.

2. Replaying Code Changes

According to Storey, diverse sources of information are available to support programmers' work in program comprehension [24]. In fact, modern IDEs including Eclipse [25] employ specific plug-ins that manage various kinds of information about code changes or human activities (see Sect. 6). A program comprehension tool could exploit such information. Tools that record fine-grained code changes and replay them [8]–[12] are typical examples of such plug-ins that have recently become feasible.

2.1 Tools

Robbes et al. proposed a toolset, SpyWare [8], that is a monitoring plug-in for IDEs. It stores the first-class changes made by a programmer on the source code of a program. These changes consist of the finest-level atomic change operations on the abstract syntax tree (AST) of the program and the higher-level composite change operations that abstract the atomic change operations. They also presented a benchmarking procedure to evaluate the change predictions based on the replay of the code changes actually recorded from IDE interactions [13].

Hattori et al. introduced a change-based approach [6] to a multi-programmer context and presented a tool, Syde [9], which supports team collaboration in multi-programmer projects. The tool records several atomic change operations on the AST of a program and two refactoring transformations performed by Eclipse. The change information is broadcasted to all team members within a project to keep them aware of what is happening in it. This awareness eases team coordination with respect to the structural conflicts in the project. Moreover, they provide evidence that their replayer allows programmers to watch past changes as they happened at the source code level. Thus, it can help programmers in various activities related to software development and program understanding [14].

Yoon and Myers developed Fluorite [10], an event-logging plug-in for Eclipse, which captures all of the low-level events in the Eclipse code editor. It stores information about each command directly invoked by a user's action (copying or pasting text, moving the cursor position, selecting text by keyboard or mouse, and undoing past commands) and each document change that contains the actual deleted and inserted text. This information enables developers or researchers to easily analyze code editing history and detect different kinds of usage patterns of the code editor. An exploratory study with Fluorite identified various reasons why developers often have to backtrack during coding [26]. Moreover, Yoon et al. presented Azurite [16], [27], a tool that employs two user interfaces to visualize fine-grained code change history: a timeline view and a code history diff view. These views provide rich manipulation of code (e.g., selective undo) and simplify answering frequently asked questions related to code editing history.

Negara et al. argued that code change data using a version control system (VCS) is inadequate for code evolution research studies [11]. Instead, such studies should leverage IDEs to capture code changes online rather than inferring them from a postmortem of the snapshots stored in the VCS. Based on this standpoint, they developed CodingTracker [11], an Eclipse plug-in that non-intrusively collects fine-grained information about the code evolution of Java programs. It records every code edit performed by programmers. Their experimental results confirmed that more detailed and accurate information is needed for understanding code changes.

In our previous work [12], we proposed OperationRecorder, an Eclipse plug-in that automatically records fine-grained changes (i.e., edit operations, not AST change operations) by continuously tracking the code edits performed on Eclipse's Java editor. Moreover, we presented OperationReplayer [15], an Eclipse plug-in that allows programmers to explore the source code restored by chronologically accumulating the recorded operations. This replayer provides functionality that not only simply replays edit operations but also backward and forward skips (i.e., rewinding and fast-forwarding) them.

These studies are all based on the assumption that deep knowledge of code changes is very useful for understanding code. This is because such changes contain explicit information about the when, how, and by whom questions that programmers confront while writing or modifying code. On the other hand, code changes do not explicitly contain information about why questions; they seem to represent how a programmer has completed the code. Therefore, replaying fine-grained code changes in chronological order can visualize past programming scenes in front of the programmers' eyes. This helps them image why they or other programmers have changed the code in the past. Such replaying provides hints of the programmers' understanding why they or others reversed undesired code changes and strongly supports conjectures about their decisions made in the past. Consequently, code change replaying tools can accelerate the reuse or the modification of existing code without much effort, although it was written by someone else, since programmers can verify the rationale behind past code changes. The replaying tools are valuable for understanding code and its changes.

2.2 Motivation

In this paper, we take OperationReplayer to reveal a problematic point concerning the use of existing replaying tools.

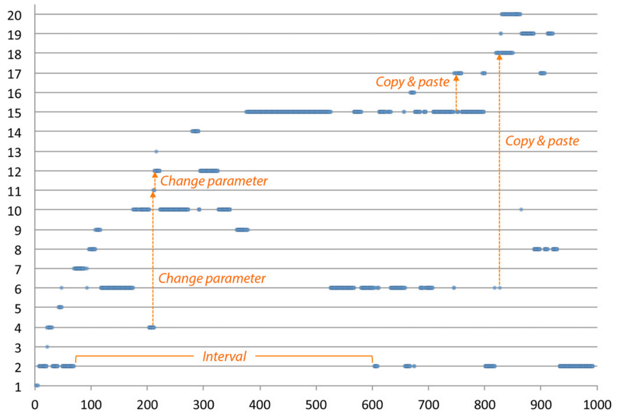


Fig. 1 Dot plot of edit operations actually recorded in our study.

Figure 1 represents the edit operations (blue dots) that were actually recorded in an exploratory study using OperationRecorder. In this study, a computer science graduate student developed a Java program for the board game *Reversi* that consists of seven source code files.

The horizontal-axis of the diagram denotes the sequence number of the edit operations (e.g., insertion, deletion, replacement of characters, and file operations) performed by the student on the Eclipse's editor to complete the file `Game.java`. The development of this file lasted about 9.5 hours. The total number of recorded edit operations for it was 996, and the file ultimately consisted of 186 lines of code. The vertical-axis denotes 20 class members (methods and fields) that appear in the file (to be precise, those identified by the syntax analysis for each of the file's 489 snapshots) during the development. For example, *number 2* corresponds to method `put(int x, int y)` of class `Game` within `Game.java`. Five methods disappeared due to their changing declarators (renaming or changing parameters). This file finally contained twelve methods and three fields.

A close look at the distribution of the actual edit operations for each of the class members reveals how much trouble replaying those edit operations gives programmers. Look at the class member labeled with *number 2*. The total number of edit operations that are directly related to it is 142. This means that 14.3% of all of the edit operations might need to be replayed when a programmer actually tries to see the past changes of this class member. However, finding only these edit operations is time-consuming with a conventional tool OperationReplayer.

To complicate matters, several intervals appear in the edit operations for each class member. This means that the construction of a class member is not restricted to a single time period. For example, consider the method labeled with *number 2* again. It has a large interval between the edit operations whose sequence numbers are 70 and 603. This interval contains the edit operations that build many other class members. In this situation, if a programmer wants to scrutinize how this class member has been created or modified, she will skip the edit operations that are unrelated to its construction by rewinding and fast-forwarding the operation history. In general, such skipping actions are annoying and interrupt thinking and concentration.

Although the result shown in Fig. 1 is an explanatory example, it is not a special. If a class could be sufficiently designed before writing its code, each class member within the class might be created at once. However, the sufficient upfront design is not always reasonable in iterative and incremental development. Adding a new field variable frequently happens and causes the concurrent update of the existing methods. Moreover, several refactoring transformations divide and/or merge them, which also cause their concurrent update. In other words, the interval of edit operations for each of the class members is inevitable.

Here, astute readers might realize that edit operations unrelated to a particular class member can be filtered out by checking its name (or signature). Unfortunately, simple filtering provided by such conventional tools as Syde [9] or CodingTracker [11] fails to address the fine-grained tracking of code changes that result from the renaming, splitting, or merging of a class member or the moving or copying of a part of its body (e.g., cut-paste or copy-paste actions). For example, the construction of the method labeled with *number 12* seems to have been performed in a relatively short period of time, with a total of 41 (9 + 32) edit operations. Simple filtering can find just these edit operations. In fact, 60 edit operations were related to the construction of this method because of its parameter changes (*number 4* → *11* → *12*). In this case, simple filtering fails to collect some edit operations that are necessary to replay the entire evolution of the method. A similar case occurred in the construction of the method labeled with *number 17* or *18* since its body absorbed several code fragments in the method labeled with *number 15* or *6* through copy-and-paste actions.

In summary, current replaying tools including OperationReplayer are convenient for detailed investigation of the construction of a particular class member if its edit operations were continuously stored in its history. However, generally, not all edit operations related to a particular class member are continuously performed. In this case, skipping actions are frequently repeated. Unfortunately, these actions are an obstacle that causes many programmers to hesitate to use existing replaying tools during comprehension tasks. Moreover, overlooking the facts of past renamings or copy and paste actions for a particular class member eliminates information about when, how, by whom it was written and modified. This increases the risk of mistakes when verifying the rationale behind past code changes. For existing replaying tools to be truly effective in realistic software development and evolution, an automatic or semi-automatic mechanism for skipping edit operations of no interest must be integrated.

3. Operation History Slicing

Our proposed slicing mechanism deals with edit operations on source code constructed under the Eclipse Java development environment. This section describes an operation history graph that represents the relationships among class members of the source code and the recorded edit operations. It also defines operation history slicing using this graph.

3.1 Recording and Replaying Edit Operations

The proposed slicing mechanism obviously assumes that all edit operations with respect to the manual and automatic code changes performed on the editor are completely collected. We adopt as a recording tool, OperationRecorder [12][†], which can automatically the record edit operations that affect the code in Eclipse's Java editor. The operations include manual typing (insertion, deletion, and replacement of text), editing by a clipboard (copying, cutting, and pasting text), undo/redo actions, and code changes by automatic transformation (code completion, quick fix, formatting, and refactoring). In addition, the operations related to a file (open, close, and save) are automatically recorded. The current version of OperationRecorder excludes the recording of actions related to file renaming and removing and ignores the detection of the renaming of classes and their members.

Each edit operation contains a particular text of code that was inserted, deleted, or replaced (i.e., deleted and inserted at the same location of the code). It also contains information about when and where the text was inserted, deleted, or replaced and by whom. Every edit operation is stored in a history file in an XML format. The following is an example of a stored edit operation:

```
<normalOperation time="1310181396607"
    dev="maru" cptype="NONE" offset="52">
  <inserted>int x, int y</inserted>
  <deleted/>
</normalOperation> .
```

Element `<normalOperation>` corresponds to one edit operation. Such edit operations have attribute `time` that stores information about when the operation was performed. It denotes the number of milliseconds since January 1, 1970. Attribute `dev` denotes the programmer's identifier (name or e-mail address, etc.). Attribute `cptype` denotes the origin of the edit operation. Its default value is `"NONE"`. If the operation was derived from a paste action, its value is `"PASTE"`. Attribute `offset` locates the starting point of the inserted, deleted, or replaced (replacing) text on the code. The inserted (or replacing) texts and deleted (or replaced) texts

[†]Version 4.5 or later was used in this paper, which is an extension of the original [12].

are enclosed as elements `<inserted>` and `<deleted>`, respectively. A copy operation `<copyOperation>` also exists. It does not affect the code appearance but is essential for knowing the origin of text inserted by a paste action.

OperationReplayer reads the data stored in the history of edit operations and restores the contents of the code of interest at a specified time. Code restoration is attained by chronologically applying every edit operation before and at the specified time to the contents of the initial code. The offset values determine the location where past edit operations were applied. Thus, programmers can see how the code has been changed as if they watch an animated movie that tracks its growth.

3.2 Operation History Graph

To collect all the edit operations that build a particular class member (method or field) of code without omissions, we must formulate the relationships between edit operations and code fragments affected by them. An *operation history graph* (OpG) is a multipartite graph that indicates which edit operation affects the code fragment(s) within a target class member.

In this paper, S_0 indicates the initial snapshot of (the contents of) the source code on which an edit operation was never performed. The subscript number is incremented by one for each edit operation that is applied. Here, p_n denotes the n-th edit operation, and S_n indicates a snapshot of the source code generated immediately after p_n was applied to its previous snapshot (S_{n-1}). In other words, S_n can be obtained after all the edit operations between p_1 and p_n are chronologically applied to S_0. A snapshot consisting of only code fragments with no syntax error is called a parseable one.

Let $M(S_i)$ be a set of all the class members within parseable snapshot S_i. If its contents are not parseable, $M(S_i)$ is empty ($M(S_i) = \emptyset$). V is a set that collects all the vertices for the class members within every snapshot and all the vertices for every edit operation. V is defined as follows:

$$\begin{aligned} V = \{\, v.m \mid m \in M(S_i) \ \wedge\ 0 \le i \le z \,\} \\ \cup \{\, v.p \mid p = p_i \ \wedge\ 1 \le i \le z \,\}, \end{aligned}$$

where i is an index number representing a subscript of a snapshot or an edit operation. z is an index number representing a subscript of latest snapshot S_z (and a subscript of latest edit operation p_z). $v.m$ denotes a vertex corresponding to class member m, and $v.p$ denotes a vertex corresponding to edit operation p.

Next, consider the edges that link two vertices included in V. We first define two adjacent snapshots S_i and S_j ($i < j$), both of which are parseable. There is no parseable snapshot S_k that satisfies $i < k < j$ since S_i and S_j are adjacent. To be precise, no snapshot exists between S_i and S_j under $j = i + 1$ or none of snapshots between S_i and S_j are parseable. The edges of an OpG G are divided into the following four types:

(a) Let p_k ($i < k \le j$) be an edit operation that changes S_i into S_j. If p_k is an edit operation and its inserted or deleted text contains any code fragment included in class member m_x within S_i, p_k affects m_x backwards. If p_k is a copy operation and its copied text is (partially) extracted from m_x, p_k also affects m_x. In these cases, $v.m_x$ ($\in V$) and $v.p_k$ ($\in V$) are linked by *backward-change* edge $v.m_x \rightarrow_b v.p_k$ in G.

(b) Consider p_k under the same situation as the aforementioned (a). If the inserted or deleted text of edit operation p_k contains any code fragment included in class member m_y within S_j, p_k affects m_y forwards. In this case, $v.p_k$ ($\in V$) and $v.m_y$ ($\in V$) are linked by *forward-change* edge $v.p_k \rightarrow_f v.m_y$ in G.

(c) Consider a situation where the contents of class member m_x within S_i remains in class member m_y within S_j without any change. In this case, $v.m_x$ ($\in V$) and $v.m_y$ ($\in V$) are linked by *no-change* (unchanged) edge $v.m_x \rightarrow_n v.m_y$ in G.

(d) Let p_x be a cut or copy operation that inserts any text into a clipboard from snapshot S_{x-1}, and let p_y be a paste operation that inserts the text stored in the clipboard into snapshot S_y. S_{x-1} and S_y may be adjacent or they may not be. If the deleted or copied text of p_x equals the inserted text of p_y after removing all white spaces, and neither cut nor copy operations were performed between p_x and p_y, $v.p_i$ ($\in V$) and $v.p_j$ ($\in V$) are linked by *ccp-change* (cut-copy-paste) edge $v.p_i \rightarrow_c v.p_j$ in G.

E is a set of all the edges that satisfy one of the above four types of edges ($\rightarrow_b$, $\rightarrow_f$, $\rightarrow_n$, or $\rightarrow_c$). An OpG is a directed graph consisting of a set of vertices (V) and a set of directed edges (E), which is represented by $G = (V, E)$.

Here, it is natural that an OpG consists of vertices representing edit operations and class members since it is used while collecting the edit operations related to the construction of a class member. On the other hand, it may not be obvious that an OpG consists of the four relationships described above. Thus, we briefly explain why an OpG involves them.

The formulation of an OpG assumes the adoption of operation history slicing. In this formulation, there are two standpoints: extracted operation history slices should include essential edit operations and should exclude nonessential ones. According to our experience, the omission of essential edit operations makes it harder for programmers to accurately capture their fine-grained code changes. Thus, operation history slices never exclude essential edit operations even if they adversely include nonessential ones.

For an OpG to satisfy this policy, its formulation utilizes a parseable snapshot and identifies the class members in it. Moreover, an edit operation between two snapshots absolutely affects (inserts, deletes, copies) their code fragments. If these code fragments belong to particular class members, the edit operation is engaged in their construction. At the same time, the contents of the class members, ex-

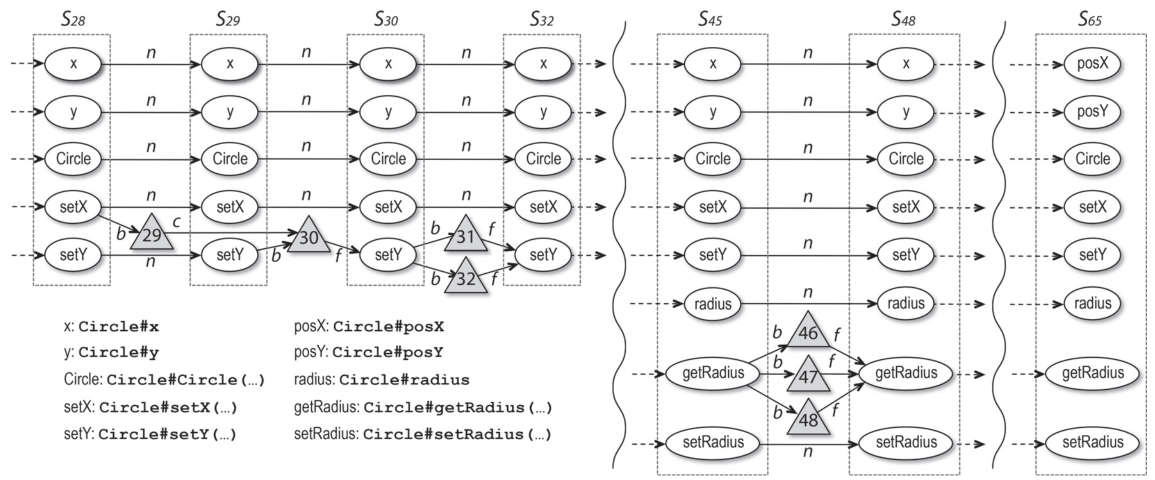


Fig. 2 An operation history graph (OpG). The whole OpG consists of 231 vertices and 240 edges.

cluding the affected code fragments, are preserved without any change made by the edit operation. This relationship is necessary to propagate the past changes of their former ones into them. Cut, copy, and paste operations affect not only the contents of the snapshots but also those of the clipboard provided by an IDE (or operation system). In the formulation, the clipboard is considered a unique and the virtual snapshot that crosses over actual snapshots. This virtual snapshot disappears from the OpG. In summary, by detecting the impacts of the edit operations on every class member in the parseable snapshots, operation history slicing does not omit any of the edit operations necessary for the construction of a particular class member.

Figure 2 depicts part of an OpG representing the construction of an explanatory example of Java code. This code contains only one class with eight class members (three fields and five methods), which was completed by 65 edit operations. The whole OpG contains all the edit operations performed during the construction and all the class members in its snapshots. Each triangle indicates a vertex corresponding to an edit operation. For example, p_{31}, p_{32}, p_{46}, p_{47}, and p_{48} are edit operations of the programmer's typing, p_{29} is a copy operation, and p_{30} is a paste operation. Each oval indicates a vertex corresponding to a class member. A dotted rectangle indicates the snapshot to which a class member belongs. Arrows marked with labels b, f, n, and c indicate the four types of edges: backward-change, forward-change, no-change edge, and ccp-change, respectively. Dashed arrows and wavy lines denote the abbreviation of edges.

3.3 Detection of Change Edges

In creating an OpG, it is necessary to detect which class member of a parseable snapshot is affected by an edit operation of interest. Our proposed mechanism tracks the offset values of the modified parts of the code and links all the edit operations with their corresponding class members. The offset value indicates the location where each edit operation was performed on the code.

Both backward-change and forward-change edges are easily detected if only one edit operation p_k is performed between adjacent and parseable snapshots S_i and S_j ($i < k \leq j$). This detection is attained by checking if the offset value of each letter in the text of p_k falls within the ranges of the offset values for class members in S_i or S_j. However, this simple detection might be violated when two or more edit operations are performed between S_i and S_j. Consider two edit operations, p_h and p_k, between S_i and S_j ($i < h < k \leq j$). If p_h shifts the offset range for m_x within S_i by inserting text before m_x and p_k deletes any text of m_x, the offset value of a letter in the deleted text of p_k indicates the wrong position on S_i. This is because this offset value indicates the position on S_h, not on S_i. Unfortunately, since S_h is not parseable, it is not present in the OpG. In this case, p_h adversely affects p_k.

To solve this problem, several offset values should be adjusted according to the length of the inserted and deleted texts of their neighboring edit operations. In the detection of backward-change and forward-change edges between an edit operation and a class member of an OpG, the adjusted offset value of the edit operation is compared with the offset range of the class member. The details of the adjustment algorithms of the offset values and the detection algorithms of the backward-change and forward-change edges will be described in Sects. 4.1 and 4.2.

After the detection of all the backward-change and forward-change edges, the no-change edges are detected. If m_x within snapshot S_i has no forward-change edge, m_y within snapshot S_j has no backward-change edge, and the full names of m_x and m_y are the same. $v.m_x$ and $v.m_y$ are linked by no-change edge $v.m_x \rightarrow_n v.m_y$. The full name is a unique identifiable name constructed by concatenating the fully qualified name of a class, special character "`#`", and the declarator (name and parameter list) of a method or the name of a field. For example, "`fqn#m(int x)`" and "`fqn#f`" are the full names for a method with declarator "`m(int x)`" and a field with variable name "`f`" of class "`fqn`", respectively. This way successfully detects the no-change edges except when a class name is changed. If a class name is changed, the detection mechanism tracks the changes of the offset range for each class member. Basically, for each p of the edit operations performed between S_i and S_j, calculation $o = o - i.p + d.p$ for offset value o of m_y is repeated. If the adjusted offset range of m_y equals the offset range of m_x and their field names or method declarators are the same, no-change edge $v.m_x \rightarrow_n v.m_y$ exists.

The detection of ccp-change edges is easy based on the definitions described in Sect. 3.2. For example, p_{29} is a copy operation and p_{30} is a paste operation. The contents of the copied text and the inserted one by paste are the same and there is no other cut or copy operation between them. Therefore, ccp-change edge $p_{29} \rightarrow_c p_{30}$ exists in the OpG shown in Fig. 2.

3.4 Operation History Slice

By traversing the vertices and edges of an OpG, the proposed mechanism can extract edit operations necessary to build a class member of interest from the history consisting of all the operations. Here, G_S is an OpG for source code S. Let $R(G_S, v.m_m)$ be a set of vertices of G_S that reach vertex $v.m_m$ corresponding to class member m within snapshot of S:

$$R(G_S, v.m) = \{\, u \in V(G_S) \mid u \rightarrow^* v.m \,\}.$$

$V(G_S)$ is a set of all the vertices of G_S. Relation $\rightarrow^*$ means the reflexive and transitive closure of relation $\rightarrow$, which indicates one of the four types of edges ($\rightarrow_b$, $\rightarrow_f$, $\rightarrow_n$, or $\rightarrow_c$) of G_S.

Here, sometimes an edit operation exists with no forward-change edge in the OpG. For example, this occurs when an edit operation deletes or cuts the entire contents of a method or a field. Since such an edit operation will be properly replayed, $R'(G_S, v.m)$ was newly derived from $R(G_S, v.m)$:

$$R'(G_S, v.m) = R(G_S, v.m) \cup \{\, w \in V(G_S) \mid u \rightarrow_b w \ \wedge \ u \in R(G_S, v.m) \,\}.$$

A reachable set of edit operations $Rp(G_S, v.m)$ is defined as follows:

$$Rp(G_S, v.m) = V_p(G_S) \cap R'(G_S, v.m).$$

$V_p(G_S)$ is a set of vertices with respect to all the edit operations.

Next consider a sequence of edit operations to be replayed. Let $Q(S)$ be a sequence that lists all the edit operations for S.

$$Q(S) = \langle\, p_1, \ \ldots, \ p_z \,\rangle.$$

For every recorded edit operation, p_1 is the first (earliest) and p_z is the last (latest). The above sequence is drawn up in chronological order. In other words, the time when p_i was performed is earlier than or equal to the time when p_j $(i < j)$ was done[†]. In this case, only one edit operation exists for S, $Q(S) = \langle p_1 \rangle$.

Operation history slice $Sq(S, m)$ is a minimal subsequence of $Q(S)$ that satisfies the following condition:

$$\forall v.p_k \in Rp(G_S, v.m)\ [\ p_k \in Sq(S, m) \wedge Sq(S, m) \sqsubseteq Q(S) \wedge \#Rp(G_S, v.m) = \#Sq(S, m)\ ].$$

$Q_1 \sqsubseteq Q_2$ means that Q_1 is a sub-sequence of Q_2. $\#Op(G_S, v.m)$ indicates the number of elements included in $Op(G_S, v.m)$, and $\#Sq(S, m)$ indicates the number of elements included in $Sq(S, m)$. These numbers are always equal. m is a slicing criterion that denotes a class member of interest within snapshot S.

The code of final snapshot S_{65} for the OpG shown in Fig. 2 has eight class members. For methods `setX()` and `setY()` within S_{65}, their operation history slices are as follows:

$$Sq(G, \texttt{setX@}S_{65}) = \langle 14, 15, 16, 17, 18, 19, 20, 21, 29, 61, 62 \rangle,$$

$$Sq(G, \texttt{setY@}S_{65}) = \langle 14, 15, 16, 17, 18, 19, 20, 21, 22, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 63, 64 \rangle.$$

Each operation history slice contains 11 or 22 edit operations. Thus, their ratios to the total number of recorded edit operations are 16.9% (= 11/65) and 33.8% (= 22/65).

4. Implementation

We developed a tool called OperationSliceReplayer that consists of four modules: Code restorer, OpG constructor, OpG slicer, and Code viewer. The Code restorer restores every snapshot of the code based on the history of edit operations recorded by OperationRecorder and then selects parseable ones from among all the restored snapshots. It employs the syntax analyzer module ASTParser built in Eclipse JDT to check whether a snapshot is parseable. In this syntax analysis, the ranges of the offset values of class members within the parseable snapshots are calculated. The OpG constructor creates an OpG by collecting vertices indicating the recorded edit operations and the class members within the restored snapshots and by detecting the edges between these vertices. The OpG slicer calculates an operation history slice for a criterion given by a programmer using the OpG. The Code viewer displays past snapshots of code in chronological order by replaying only the edit operations included in the slice.

This section first describes two algorithms that adjust the offset values of the letters stored in each edit operation and two algorithms of the backward-change and forward-change edges using the adjusted offset values. These algorithms are all implemented in the OpG constructor. This section also explains the usage of OperationSliceReplayer.

[†]Although two edit operations are performed at the same time, OperationRecorder suitably determines their chronological order and records them in XML files.

4.1 Adjustment Algorithms of Offset Values

Algorithms 1 and 2 explain the procedures that adjust the offset values to detect the backward-change and forward-change edges. Here, the offset value of the n-th letter in the text of p is represented by $o.p[n]$. $i.p$ denotes the length of the inserted text of p, and $d.p$ denotes the length of the deleted text of p. Function `GetOps`(a, b), which returns an ordered collection of the edit operations between edit operations p_a and p_b, is extracted from the original sequence that lists all the edit operations in chronological order. In `GetOps`(a, b), the first one is p_a and the last one is p_b. Function `GetOpsRev`(a, b) returns an ordered collection that lists the edit operations in `GetOps`(a, b) in reverse order.

Here, we show examples of the adjustment of the offset values using the snapshots and edit operations in Fig. 3. In parseable snapshots S_{45} and S_{48}, `[\t]` and `[\n]` are invisible symbols indicating a tab character and a new-line one, respectively. Edit operation p_{46} for S_{45} adversely affects edit operation p_{47} since offset value $o.p_{46}[1]$ (= 293) was less than or equal to $o.p_{47}[1]$ (= 303). Similarly, both p_{46} and p_{47} adversely affect edit operation p_{48}. In these cases, the adjustment for $o.p_{47}[1]$ and $o.p_{48}[1]$ should be done based on Algorithm 1. For example, with respect to the first letter in the text of p_{47} for S_{45}, adjusted offset value $o.p'_{47}[1]$ was calculated as follows:

$$\begin{aligned} o.p'_{47}[1] &= o.p_{47}[1] - i.p_{46} + d.p_{46} \\ &= 303 - 11 + 0 = 292. \end{aligned}$$

Note however, that the value of $o.p'_{47}$ finally became equal to the value of $o.p_{46}$, which was $o.p'_{47} = 293$, since the actual value of $o.p'_{47}$ was less than $o.p_{46}$. In another example with respect to the first letter in the text of p_{48} for S_{45}, adjusted offset value $o.p'_{48}[1]$ was calculated as follows:

$$\begin{aligned} o.p'_{48}[1] &= o.p_{48}[1] - i.p_{47} + d.p_{47} - i.p_{46} + d.p_{46} \\ &= 303 - 0 + 1 - 11 + 0 = 293. \end{aligned}$$

The above calculations are both related to the edit operations for their precedent parseable snapshot. When the edit operation must be adjusted to its subsequent parseable snapshot, the addition and subtraction for its offset value must be interchanged, as shown in Algorithm 2. However, this adjustment was not required for p_{46} and p_{47} in Fig. 3 since $o.p_{46}[n_1] \le o.p_{47}[n_2] \le o.p_{47}[n_3]$ for any letter $(1 \le n_1 \le i.p_{46} \land 1 \le n_2 \le d.p_{47} \land 1 \le n_3 \le i.p_{48})$ in the text of p_{46}, p_{47}, and p_{48}.

Algorithm 1: *AdjustBackwardOffset*

```
in  : $o$ – offset value of a letter in a target edit operation
out: $o'$ – offset value after adjustment
$o' = o$
foreach $p \in$ GetOpsRev($i + 1, k - 1$) do
    if $o.p[1] \le o'$ then
        $o' = o' - i.p + d.p$
        if $o' < o.p[1]$ then $o' = o.p[1]$
    end
end
return $o'$
```

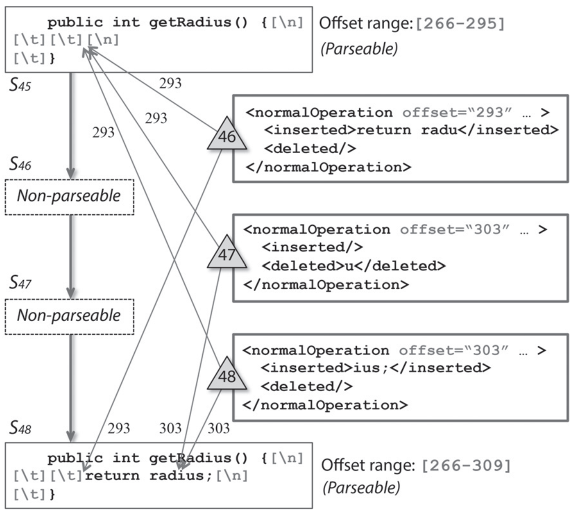
Algorithm 2: *AdjustForwardOffset*

```
in  : $o$ – offset value of a letter in a target edit operation
out: $o'$ – offset value after adjustment
$o' = o$
foreach $p \in$ GetOps($k + 1, j$) do
    if $o.p[1] \le o'$ then
        $o' = o' + i.p - d.p$
        if $o' < o.p[1]$ then $o' = o.p[1]$
    end
end
return $o'$
```

4.2 Detection Algorithms of Change Edges

Algorithms 3 and 4 explain the procedures that detect the backward-change and forward-change edges. Here, o' is the adjusted offset value of a letter in the text of edit operation p. If the value of o' falls within the range of offset values $[o_s, o_e]$ of class member m in a previous parseable snapshot, that is, $o_s \le o' \le o_e$, backward-change edge $v.m \rightarrow_b v.p$ is added to an OpG. o_s and o_e are the offset values of the starting and ending points of m. For a class member in a subsequent parseable snapshot, the same comparison is done to detect forward-edit edge $v.p \rightarrow_f v.m$. Since p potentially affects multiple class members at the same time, this offset comparison is repeatedly applied to every letter in the text. To be precise, each letter in the deleted or copied text is compared to detect a backward-change edge, and each letter in the inserted text is compared to detect a forward-change edge.

In Fig. 3, the adjusted offset value of first letter "`r`" of the text that was inserted by p_{46} equals 293 (i.e., $o.p'_{46}[1]$



```
    public int getRadius() {[\n]
[\t][\t][\n]
[\t]}
```

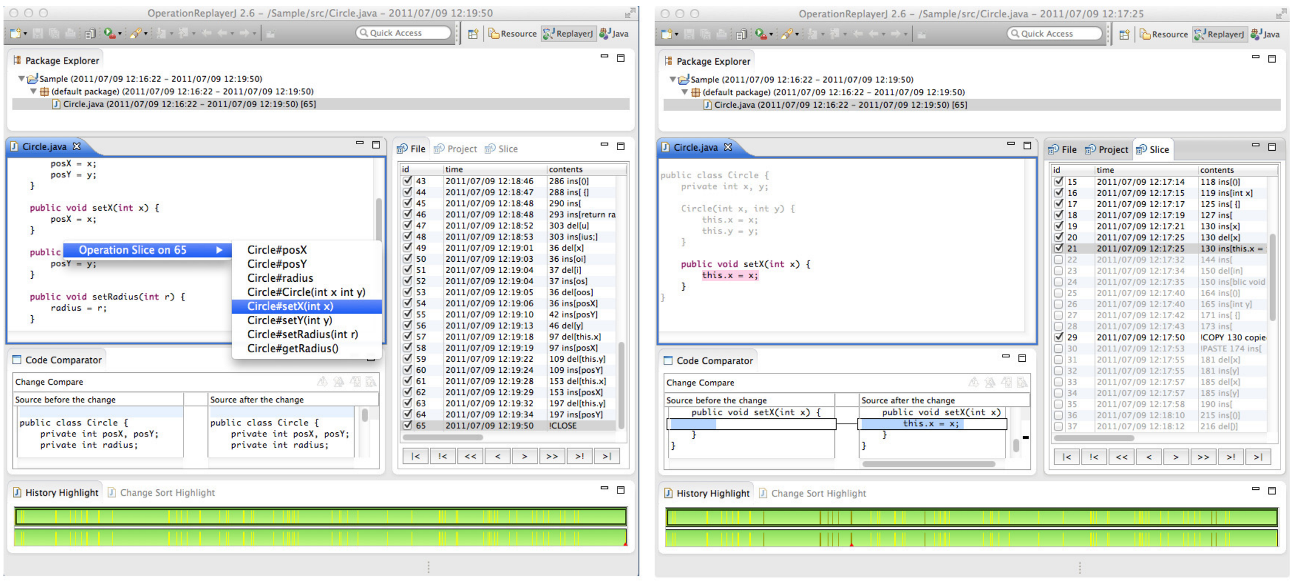
```
<normalOperation offset="293" … >
  <inserted>return radu</inserted>
  <deleted/>
</normalOperation>
```

```
<normalOperation offset="303" … >
  <inserted/>
  <deleted>u</deleted>
</normalOperation>
```

```
<normalOperation offset="303" … >
  <inserted>ius;</inserted>
  <deleted/>
</normalOperation>
```

```
    public int getRadius() {[\n]
[\t][\t]return radius;[\n]
[\t]}
```

Fig. 3 Detection of backward- and forward-change edges.



```
Sample (2011/07/09 12:16:22 – 2011/07/09 12:19:50)
  (default package) (2011/07/09 12:16:22 – 2011/07/09 12:19:50)
    Circle.java (2011/07/09 12:16:22 – 2011/07/09 12:19:50) [65]
```

```
        posX = x;
        posY = y;
    }

    public void setX(int x) {
        posX = x;
    }

    public
        posY = y;
    }

    public void setRadius(int r) {
        radius = r;
    }
```

Source before the change

```
public class Circle {
    private int posX, posY;
    private int radius;
```

Source after the change

```
public class Circle {
    private int posX, posY;
    private int radius;
```

id	time	contents
✓ 43	2011/07/09 12:18:46	286 ins[()]
✓ 44	2011/07/09 12:18:47	288 ins[{]
✓ 45	2011/07/09 12:18:48	290 ins[
✓ 46	2011/07/09 12:18:48	293 ins[return ra
✓ 47	2011/07/09 12:18:52	303 del[u]
✓ 48	2011/07/09 12:18:53	303 ins[ius;]
✓ 49	2011/07/09 12:19:01	36 del[x]
✓ 50	2011/07/09 12:19:03	36 ins[oi]
✓ 51	2011/07/09 12:19:04	37 del[i]
✓ 52	2011/07/09 12:19:04	37 ins[os]
✓ 53	2011/07/09 12:19:05	36 del[oos]
✓ 54	2011/07/09 12:19:06	36 ins[posX]
✓ 55	2011/07/09 12:19:10	42 ins[posY]
✓ 56	2011/07/09 12:19:13	46 del[y]
✓ 57	2011/07/09 12:19:18	97 del[this.x]
✓ 58	2011/07/09 12:19:19	97 ins[posX]
✓ 59	2011/07/09 12:19:22	109 del[this.y]
✓ 60	2011/07/09 12:19:24	109 ins[posY]
✓ 61	2011/07/09 12:19:28	153 del[this.x]
✓ 62	2011/07/09 12:19:29	153 ins[posX]
✓ 63	2011/07/09 12:19:32	197 del[this.y]
✓ 64	2011/07/09 12:19:34	197 ins[posY]
✓ 65	2011/07/09 12:19:50	!CLOSE

(a) Selecting the slicing criterion

```
Sample (2011/07/09 12:16:22 – 2011/07/09 12:19:50)
  (default package) (2011/07/09 12:16:22 – 2011/07/09 12:19:50)
    Circle.java (2011/07/09 12:16:22 – 2011/07/09 12:19:50) [65]
```

```
public class Circle {
    private int x, y;

    Circle(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public void setX(int x) {
        this.x = x;
    }
}
```

Source before the change

```
    public void setX(int x) {

    }
}
```

Source after the change

```
    public void setX(int x)
        this.x = x;
    }
}
```

id	time	contents
✓ 15	2011/07/09 12:17:14	118 ins[()]
✓ 16	2011/07/09 12:17:15	119 ins[int x]
✓ 17	2011/07/09 12:17:17	125 ins[{]
✓ 18	2011/07/09 12:17:19	127 ins[
✓ 19	2011/07/09 12:17:21	130 ins[x]
✓ 20	2011/07/09 12:17:25	130 del[x]
✓ 21	2011/07/09 12:17:25	130 ins[this.x =
☐ 22	2011/07/09 12:17:32	144 ins[
☐ 23	2011/07/09 12:17:34	150 del[in]
☐ 24	2011/07/09 12:17:35	150 ins[blic void
☐ 25	2011/07/09 12:17:40	164 ins[()]
☐ 26	2011/07/09 12:17:40	165 ins[int y]
☐ 27	2011/07/09 12:17:42	171 ins[{]
☐ 28	2011/07/09 12:17:43	173 ins[
✓ 29	2011/07/09 12:17:50	!COPY 130 copie
☐ 30	2011/07/09 12:17:53	!PASTE 174 ins[
☐ 31	2011/07/09 12:17:55	181 del[x]
☐ 32	2011/07/09 12:17:55	181 ins[y]
☐ 33	2011/07/09 12:17:57	185 del[x]
☐ 34	2011/07/09 12:17:57	185 ins[y]
☐ 35	2011/07/09 12:17:58	190 ins[
☐ 36	2011/07/09 12:18:10	215 ins[()]
☐ 37	2011/07/09 12:18:12	216 del[)]

(b) Displaying the operation history slice

Fig. 4 OperationSliceReplayer perspective in Eclipse.

Algorithm 3: *DetectBackwardChangeEdges*

```
in   : $p$ – target edit operation
in   : $M$ – set of class members in target snapshot
inout: $G$ – OpG
for $m \in M$ do
    $len \leftarrow$ length of deleted or copied text of $p$
    for $n \leftarrow 1$ to $len$ do
        $o'$ = AdjustBackwardOffset($o.p[n]$)
        if $o'$ falls within offset range for $m$ then
            Adds backward-change edge $m \rightarrow_b p$ into $G$
        end
    end
    $len \leftarrow$ length of inserted text of $p$
    if $len \neq 0$ then
        $o'$ = AdjustBackwardOffset($o.p[1]$)
        if $o'$ falls within offset range for $m$ then
            Adds backward-change edge $m \rightarrow_b p$ into $G$
        end
    end
end
```

Algorithm 4: *DetectForwardChangeEdges*

```
in   : $p$ – target edit operation
in   : $M$ – set of class members in target snapshot
inout: $G$ – OpG
if $p$ is a copy operation then return

for $m \in M$ do
    $len \leftarrow$ length of inserted text of $p$
    for $n \leftarrow 1$ to $len$ do
        $o'$ = AdjustForwardOffset($o.p[n]$)
        if $o'$ falls within offset range for $m$ then
            Adds forward-change edge $m \rightarrow_f p$ into $G$
        end
    end
    $len \leftarrow$ length of deleted text of $p$
    if $len \neq 0$ then
        $o'$ = AdjustForwardOffset($o.p[1]$)
        if $o'$ falls within offset range for $m$ then
            Adds forward-change edge $m \rightarrow_f p$ into $G$
        end
    end
end
```

$= o.p_{46}[1] = 293$). The range of the offset values of method `getRadius()` within S_{45} is [266, 295]. Thus, letter "`r`" in the inserted text of p_{46} was probably inserted into `getRadius()` within S_{45}. Next consider p_{47}. In this case, S_{46} is not parseable, and thus a class member corresponding to p_{47} cannot be detected in S_{46}. Moreover, the offset value of a letter in the text of p_{47} must be adjusted. Since the adjusted offset value $o.p'_{47}[1]$ (= 293) falls within the range [266, 295] of `getRadius()` within S_{45}, the first letter "`u`" in the deleted text of p_{47} was probably deleted from `getRadius()` of S_{45}. In the same manner, the first letter "`i`" in the inserted text of p_{48} was considered to be inserted into `getRadius()` of S_{45}. Consequently, three backward-change edges, $v.m_x \rightarrow_b v.p_{46}$, $v.m_x \rightarrow_b v.p_{47}$, and $v.m_x \rightarrow_b v.p_{48}$, were detected, where $v.m_x$ indicates the vertex for `getRadius()` of S_{45}. Similarly, three forward-change edges, $p_{46} \rightarrow_f v.m_y$, $p_{47} \rightarrow_f v.m_y$, and $p_{48} \rightarrow_f v.m_y$, were detected, where $v.m_y$ indicates the vertex for `getRadius()` of S_{48}. These six change edges can be seen in Fig. 2.

4.3 Usage

Since OperationSliceReplayer was built as an Eclipse perspective, a programmer can instantly switch from its Java development perspective. Figure 4 shows screenshots of the OperationSliceReplayer perspective on Eclipse. The view at the top helps a programmer select code (a file) to be replayed. The code is restored in the left view. To see the growth of a particular class member, she can activate the slicing menu, as shown in the left screenshot. This menu presents all the class members in the currently restored code that is parseable. If the code is not parseable, the tool tries to find an immediately precedent parseable snapshot and presents a list of its class members on the menu. The recorded edit operations are listed in the right view. She can replay one-by-one (plus rewind and fast-forward) the whole history of the recorded edit operations by pushing each of the buttons. In the right screenshot, the edit operations and the code fragments included in the operation history slice for the criterion given by the programmer are displayed in black. On the other hand, the edit operations and code fragments not included in the slice are displayed in gray. She can replay only the sliced history of the edit operations.

Here we briefly describe how to restore the code related to only the edit operations included in an operation history slice. Actually, the partial replay of all the recorded edit operations often hampers the correct restoration of the original snapshot since such replay often revokes the change of the offset values performed by the edit operations that are not included in the slice. This might penalize programmers' tasks for understanding code. Thus, it is significant in the implementation of OperationSliceReplayer to correctly restore the original snapshot without changing its layout. In other words, the apparent positions of all the class members within the original snapshot must be preserved, although the recorded edit operations are partially replayed. To realize such replay, the tool replays all the recorded the edit operations without removing any of them. It preserves the original color of the code fragments corresponding to the edit operations included in an operation history slice, but decolors the code fragments corresponding to the edit operations not included in the slice. This coloring helps programmers distinguish the two types of edit operations that are either necessary or unnecessary to build a particular class member.

5. Evaluation

We carried out experiments with OperationSliceReplayer to assess its effectiveness. These experiments dealt with a situation where a programmer wants to understand the evolution of a particular class member within an existing Java program. If she can obtain the history of the code of the program, she can usually exploit it with the replaying tools mentioned in Sect. 2.1. OperationSliceReplayer provides the functionality of replaying past edit operations as well as these conventional tools. In addition, it reduces the search space for investigating past code changes by automatically selecting only the edit operations that can help her understanding.

To develop research questions, we sorted out the following five techniques when she understands past code changes:

T1 She investigates every edit operation stored in the operation history for code appearing in all the files.

T2 She investigates the edit operations related to a file containing a class member of interest.

T3 She investigates the edit operations related to the construction of a class member of interest by slicing the whole operation history for the code appearing in all files (inter-files mode in OperationSliceReplayer).

T4 She investigates the edit operations related to the construction of a class member of interest by slicing the operation history for the code appearing in a file containing the class member (intra-file mode in OperationSliceReplayer).

T5 Several conventional tools including Syde [9] or CodingTracker [11] infer the corresponding AST nodes from the collected raw edits. By using these tools, she investigates only the edit operations filtered with respect to a class member of interest.

Based on these techniques, we address the following research questions:

RQ1 How much does the automation of operation history slicing reduce the effort to investigate code changes when a programmer understands them?

RQ2 Are the outcomes of operation filtering sufficient to present an accurate replay, compared with those of operation history slicing?

Figure 5 depicts the edit operations stored in the operation history. The five techniques deal with their respective areas enclosed in dotted or broken lines.

T1 obtains all of the information about code changes but requires a huge amount of effort. Therefore, T1 is of no practical use. On the other hand, T2 fails to obtain enough information, unlike T1. Nevertheless, it is preferable except for T3 and T4, since any class member is enclosed in a file in a Java program. T3 conceptually provides enough information like T1 (if an operation history slicing algorithm and its implementation are perfect) to restage past code changes on a particular class member. Similarly, T4 provides the same information as T2 in restaging. From these perspectives, we answer RQ1.

Whereas RQ1 focuses on effort reduction by operation history slicing, RQ2 assesses the accuracy of operation filtering and slicing. Both T3 (slicing) and T5 (filtering) have

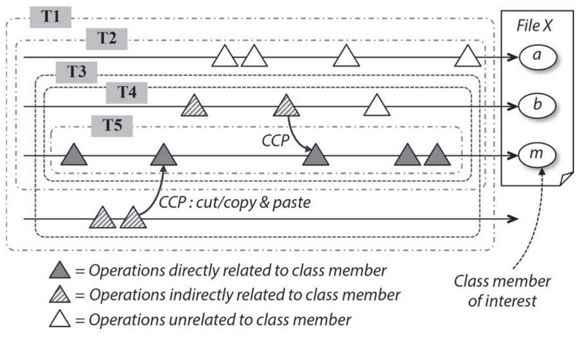

T1
T2
T3
T4
T5
CCP
CCP : cut/copy & paste
File X
a
b
m
= Operations directly related to class member
= Operations indirectly related to class member
= Operations unrelated to class member
Class member of interest

Fig. 5 Edit operations used for respective techniques.

Table 1 Main characteristics of experimental Java application programs.

Application	Development period	Files	Lines of code	Class members	Edit operations	Parseable snapshots
App 1	2008/11/05 ~ 2008/11/06	7	500	59	2,650	1,248
App 2	2012/08/10 ~ 2012/08/31	7	772	88	3,717	1,688
App 3	2013/10/03 ~ 2013/11/29	14	1,831	215	16,204	9,976

Table 2 Information about operation history graphs for application programs.

Application	$\#V_p$	$\#V_m$	$\#V$	$\#E_b$	$\#E_f$	$\#E_n$	$\#E_c$	$\#E$
App 1	2,650	15,459	18,109	2,279	2,452	14,384	72	19,187
App 2	3,714	25,214	28,928	3,089	3,420	23,726	52	30,287
App 3	16,204	173,052	189,256	13,869	14,284	164,657	510	193,320

a beneficial effect on the reduction of the search space for investigating edit operations. In this situation, if the accuracy of T5 closely resembles that of T3, T3 is unneeded. Conversely, if T5 overlooks several edit operations and fails to accurately restage past code changes, T3 appreciates in value. From this perspective, we answer RQ2.

We know that readers want to learn the practical benefits to programmers who have to understand how existing code has been written and modified. However, it is hard to determine their tasks that are performed in the actual software development and evolution and that directly assess such benefits. Therefore, our experiments assume that the automatic reduction of search space eliminates the annoying tasks that will likely happen while understanding the existing code. Moreover, the accurate detection of the search space mitigates the risk of misunderstanding. This elimination and mitigation help programmers easily and quickly obtain accurate knowledge about the code.

5.1 Experiment Design

In our experiments, we prepared a set of edit operations that were actually collected during the development of Java application programs by three computer science graduate students. Table 1 summarizes the main characteristics: the numbers of lines of code of the final snapshot, the class members appearing in the final snapshot, the edit operations recorded during development, and the parseable snapshots restored from the edit operations of each program used in the experiments. Both App 1 and App 2 implement a board game, and App 3 implements a puzzle video game. Unfortunately, the original history of the edit operations contained several deficiencies. Thus, we manually fixed them to replay every edit operation without any conflicts[†]. The values shown in Table 1 correspond to the operation history after fixing.

Table 2 shows information about the operation history graphs for the application programs. Columns $\#V_p$ and $\#V_m$ indicate the numbers of vertices that correspond to all the edit operations and to all the class members appearing in every parseable snapshot, respectively. The value of $\#V$ denotes the total number of vertices of an operation history graph ($\#V = \#V_p + \#V_p$). Columns $\#E_b$, $\#E_f$, $\#E_n$, and $\#V_c$ indicate the numbers of backward-change, forward-change, no-change, and ccp-change edges, respectively. The value of $\#E$ denotes the total number of edges of an operation history graph ($\#E = \#E_b + \#E_f + \#E_n + \#E_c$).

We obtained the operation history slices in the following two ways:

Exp 1 We selected all the class members that appear in the final snapshot of each of the application programs as the operation history slicing criteria. As a result, we totally obtained 362 (59 + 88 + 215) slices from all the programs.

Exp 2 We randomly selected 1,000 class members that appear in every snapshot as the operation history slicing criteria. As a result, we obtained 1,000 slices (6.4%, 4.0%, and 0.58%) from the respective application programs.

5.2 Experimental Results

Table 3 summarizes the results with experiment Exp 1. LOC and #M show the numbers of lines of code and the class members for each file. $T2_{Ave}$, $T3_{Ave}$, $T4_{Ave}$, and $T5_{Ave}$ denote the average number of edit operations for each file, which are collected by T2, T3, T4, and T5, respectively. S_{Ave}, W_{Ave}, and F_{Ave} will be explained in Sects. 5.2.1 and 5.2.2.

5.2.1 Reduction of Effort

To answer RQ1, we prepared two evaluation metrics. One represents the ratio of the number of edit operations that programmers investigate to the number of edit operations that they ignore. This metric is called the size ratio. In our experiments, we measured the size ratio as follows:

$$S(m) = T4(m)\ /\ T2(m).$$

$T2(m)$ or $T4(m)$ indicates the number of edit operations for class member m, which is obtained by T2 or T4. In this metric, we set up a baseline by T2 instead of T1 since a programmer seldom uses T1 and tends to use T2. Every value of $S(m)$ is between 0 and 1 (0%~100%). The lower

[†]OperationSliceReplayer detects several conflicts such as the mismatch between texts of the code restored from edit operations and the code when saving it, the deletion of non-existent text, and the insertion of text out of the code.

Table 3 Experimental results with Exp 1.

Application	File	LOC	#M	$T2_{Ave}$	$T3_{Ave}$	$T4_{Ave}$	$T5_{Ave}$	S_{Ave}	W_{Ave}	F_{Ave}
App 1	Board.java	40	5	310	67.0	67.0	22.6	21.6%	2.0	35.9%
	CUIView.java	39	3	225	99.7	71.7	69.3	31.9%	1.8	61.1%
	Game.java	186	15	996	108.5	108.5	60.0	10.9%	4.4	79.0%
	Stone.java	29	7	122	18.3	18.3	14.9	15.0%	4.5	73.8%
	SwingView.java	190	27	865	63.8	53.1	20.5	6.1%	19.8	61.4%
	View.java	5	1	11	5.0	5.0	5.0	45.5%	1.6	100.0%
	Reversi.java*	11	1	121	115.0	115.0	115.0	95.0%	1.0	100.0%
App 2	Board.java	352	34	1,692	55.9	55.9	37.7	3.3%	42.0	72.2%
	BoardPanel.java	138	17	680	39.4	39.4	37.2	5.8%	7.7	93.4%
	ControlPanel.java	52	9	179	53.3	53.3	9.9	29.8%	1.2	61.7%
	InfoPanel.java	40	7	175	21.6	21.6	21.6	12.3%	1.4	100.0%
	Main.java	16	1	533	516.0	485.0	428.0	91.0%	1.1	82.9%
	Reversi.java	87	12	90	374.5	14.5	8.9	16.1%	2.2	26.6%
	Xoard.java**	87	6	365	59.3	59.3	58.2	16.3%	2.9	97.5%
App 3	ComPanel.java	15	2	158	45.5	39.0	18.5	24.7%	2.7	54.2%
	Field.java	389	35	3,805	239.3	231.1	70.3	6.1%	75.8	59.0%
	GameFrame.java	81	2	796	374.5	374.5	364.5	47.0%	1.9	52.4%
	GaugePanel.java	98	13	541	68.5	45.0	33.8	8.3%	7.5	45.0%
	KaeruPanel.java	174	10	2,096	589.1	353.9	192.5	16.9%	15.4	61.0%
	NextPuyoPanel.java	45	7	248	27.1	27.1	19.6	10.9%	2.7	79.5%
	OjamaField.java	77	18	203	65.2	10.4	7.4	5.1%	23.5	42.5%
	OjamaPanel.java	192	36	1,548	149.0	47.3	18.7	3.1%	19.0	40.5%
	PlayerPanel.java	79	4	755	471.5	167.8	148.3	22.2%	20.8	27.4%
	Puyo.java	22	8	174	12.6	12.6	8.9	7.3%	2.1	63.8%
	PuyoPair.java	189	24	1,223	101.9	86.2	37.5	7.0%	11.6	55.0%
	PuyoPanel.java	400	46	4,288	194.6	167.0	66.8	3.9%	43.8	67.6%
	ScorePanel.java	61	95	332	32.0	32.0	25.9	9.6%	6.3	73.6%
	TempField.java	9	1	37	18.0	18.0	18.0	48.6%	1.1	100.0%

*The original file name was replaced with Reversi.java for privacy protection.
**The file name was changed from Board.java since OperationSliceReplayer cannot deal with file renaming.

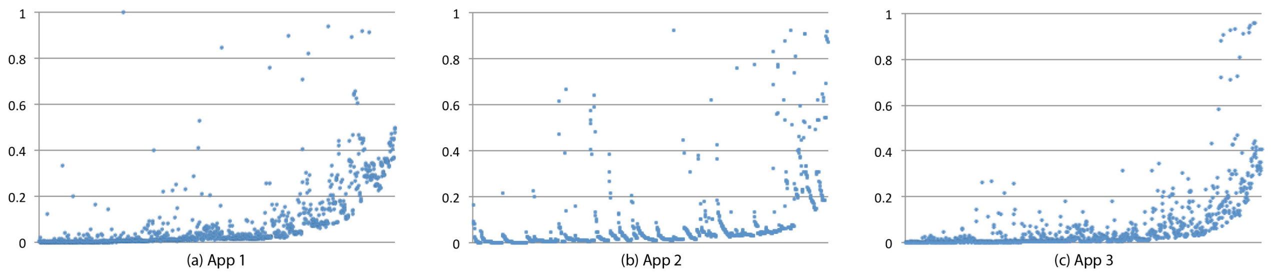


Fig. 6 Size ratio for each operation history slice (S).

the value of $S(m)$ is, the higher is the rate of the reduction of edit operations that programmers care about.

The other metric represents the spreading of edit operations. This is called the spreading ratio. Although operation history slicing leads to various rates of the reduction of the number of edit operations, its impact depends on how the edit operations are spread in a slice. As mentioned in Sect. 2.2, if edit operations disperse in the wide range of the operation history, it is hard for programmers to find them. In addition, they often perform skipping actions for the replaying the operations. Accordingly, the burden on programmers is increased. Conversely, if edit operations are concentrated, programmers can easily replay them. In the experiments, we measured the spreading ratio in an operation history slice as follows:

$$W(m) = Range(first(T4(m)), last(T4(m))) \ / \ T4(m).$$

$first(T4(m))$ and $last(T4(m))$ indicate the first and last edit operations in the chronological sequence of all the edit operations in $T4(m)$. $Range(o_f, o_l)$ denotes the number of edit operations sandwiched between o_f and o_l, which are extracted from the whole operation history. The higher the value of $W(m)$ is, the wider is the range of edit operations that programmers care about.

In Table 3, S_{Ave} denotes the average value of all size ratios $S(m)$ for each file in Exp 1. A low value of S_{Ave} represents a high reduction of required effort of programmers with respect to the slice size (the number of edit operations in a slice). Figure 6 shows more generalized results with Exp 2[†]. These results present the overall trend of $S(m)$. The

[†]There seems to be some correlation between slice size and size ratio. The values of the Spearman's rank correlation coefficients are 0.8671387, 0.7728087, 0.8234806 for App 1, App 2, and App 3, and their p-values for the test of no correlation are all less than 0.05.

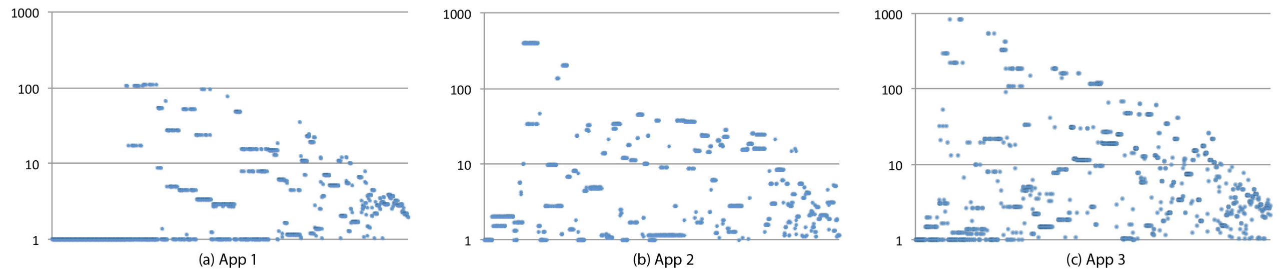


Fig. 7 Spreading ratio for each operation history slice (W).

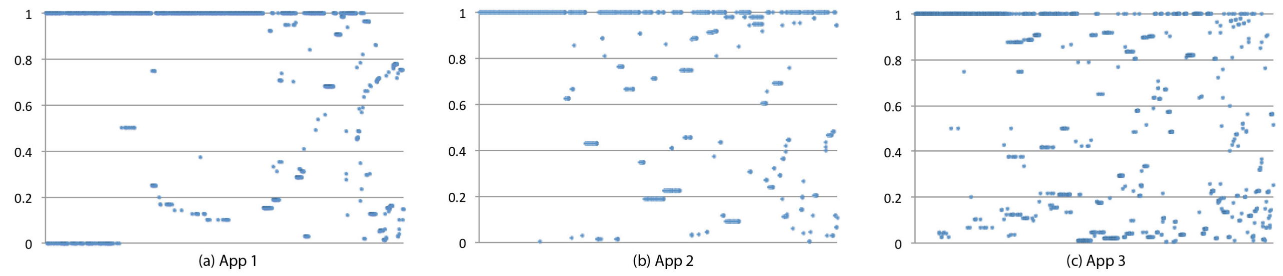


Fig. 8 Accuracy ratio for each operation history slice (F).

horizontal-axis of the three diagrams denotes slices that are arranged in the ascending order by size. The vertical-axis denotes the values of $S(m)$. For every application, most of the values of $S(m)$ are low. Small-sized slices greatly reduce the required effort. One result, for example, is that the values for 984, 953, and 985 of all the 1,000 slices are less than 0.5 (50%). This means that more than half of the edit operations is not needed to be replayed for most of the class members.

In Table 3, W_{Ave} denotes the average value of all the spreading ratios $W(m)$ for each file in Exp 1. All the values of $W(m)$ are between 1 and 75.8. If the value of W_{Ave} is close to 1, edit operations obtained by operation history slicing are concentrated in a narrow range. In this case, we cannot expect to reduce the burden on programmers. Conversely, a high value of W_{Ave} greatly reduces the burden. According to the results in Exp 1, we expect to reduce the burden for several files with high values. Similar results were derived from Exp 2 as shown in Fig. 7. The horizontal-axis of the three diagrams denotes slices that are also shown in Fig. 6. The vertical-axis denotes the values of $W(m)$. For every application, a considerable number of slices are likely to greatly reduce the burden. As an example, the values of $W(m)$ for 174, 315, and 344 slices exceed 10. The edit operations to be replayed in these slices disperse in a range more than tenfold their sizes.

In summary, the experimental results with respect to $S(m)$ demonstrate that the automation of operation history slicing of OperationSliceReplayer can reduce the required effort to investigate code changes. Moreover, the experimental results with respect to $W(m)$ reveal that troublesome actions can be removed for skipping edit operations that do not have to be replayed.

5.2.2 Accuracy

To answer RQ2, we prepared one evaluation metric that represents the ratio of the number of edit operations obtained by operation history slicing (T3) to the number of edit operations obtained by filtering (T5). In the experiments, this ratio, which is called the accuracy ratio by filtering, is calculated as follows:

$$F(m) = T5(m) \;/\; T3(m).$$

$T3(m)$ or $T5(m)$ indicates the number of edit operations for class member m, which is obtained by T3 or T5. Here, the slicing algorithm of OperationSliceReplayer embraces an algorithm that infers the correspondence of the class members to the edit operations and extracts a slice that always contains the edit operations collected by filtering. Every value of $F(m)$ is between 0 and 1 (0%~100%). If the value of $F(m)$ equals 1, filtering T5 selects every edit operation related to the construction of m. Conversely, if the value of $F(m)$ is less than 1, an edit operation was overlooked by T5. The lower the value of $F(m)$ is, the more accurate slicing is compared with filtering.

In Table 3, F_{Ave} denotes the average values of all the accuracy ratios $F(m)$ for each file in Exp 1. Figure 8 presents an overall trend of $F(m)$ with Exp 2. The horizontal-axis of the three diagrams denotes the slices also shown in Fig. 6. The vertical-axis denotes the values of $F(m)$. For example, one result is that the values for 388, 424, and 612 slices do not equal 1 (100%). In other words, operation filtering cannot sufficiently pick up the edit operations to restage the entire construction of a particular class member in 38.8%, 42.4%, and 61.2% cases. These results demonstrate that operation filtering collects the edit operations that are insufficient to capture accurate code changes.

5.3 Discussion

This section discusses several factors that can alter the effectiveness of OperationSliceReplayer.

5.3.1 Correctness of Slices

In respect to the correctness of the extracted operation history slices, we manually verified snapshots of code restored by replaying the edit operations included in the slices. Operation history slicing is expected to inherit a property from program slicing [22]. In other words, the original contents of a target class member on an arbitrary snapshot should always be the same as the contents restored by replaying only the edit operations included in an operation history slice for the same target class member on the same snapshot. As far as we examined several operation history slices in the experiments, all the slices satisfied this property. For example, OperationSliceReplayer tracked the renaming of a class member and correctly restored it without overlooking any edit operations related to a class member that later changed into the renamed class member.

However, our examination revealed the shortcomings of operation history slicing at the same time. In creating an OpG, our proposed mechanism tries to detect the backward-edit and forward-edit edges that represent the impact of the edit operations on the class members by comparing their offset values. This comparison can find the changes that were performed inside each of the class members but ignores those performed outside of it. For this reason, documentation comments (e.g., Javadoc comments in most cases) that immediately precede class member declarations are not included in any operation history slice. In addition, if the body of a method is divided into multiple smaller methods, every edit operation that builds the original method is included together in a slice for either one of the divided methods. In this case, a programmer reluctantly replays several edit operations in which she has no interest. To construct more accurate slices, further investigation of fine-grained code changes is required, although useful techniques that can improve our proposed mechanism remain unclear.

5.3.2 Performance of Implementation

All the experiments were carried out on a MacBook Pro with an Intel Core i7 3.1GHz CPU, running on Mac OS X 10.10.4 and Eclipse 4.3.1 loaded with a Java VM (JRE 1.7.0) to which 4GB of memory was allocated. The processing times to extract the operation history slices in OperationSliceReplayer were about 8, 7, and 60 seconds for App 1, App 2, and App 3 in Exp 1, respectively. The creation of an OpG in the OpG constructor required the most time. The processing times to calculate an operation history slice in the OpG slicer were all less than 1 second. The same results were observed in Exp 2. This is because the current implementation of OperationSliceReplayer creates an OpG the first time and holds information about all the nodes and edges in the memory.

According to the results of the experiments, if a target application is built by a series of 2,000 edit operations (its size resembles that of App 3), the current implementation has potential to complete the extraction of any operation history slice in an acceptable time period for practical use. However, this might be a special case. A huge amount of edit operations are expected to be recorded under large-scale and/or long-term projects in actual software development and evolution. To work around this realistic case, the implementation needs to be improved in terms of speed and space efficiency.

One challenging solution is an improved implementation that incrementally both creates an OpG every time a snapshot of source code becomes parseable and stores its information in the external memory. Fortunately, not every edit operation recorded in the past will be canceled and a new one will simply be added. Moreover, the new implementation independently creates part of an OpG by analyzing edit operations between two parseable snapshots. In other words, it only requires edit operations that were performed after the immediately precedent parseable snapshot. This allows it to flush out information about edit operations that were already analyzed.

Once a programmer activates operation history slicing, the new implementation loads information about the OpG and restores it. In this case, it does not need to load all of the information. The current implementation has already managed the OpG in a hierarchical way. It consists of sub-OpGs for respective files and ccp-change edges across them since the change edges (except the ccp-change ones) can connect only the nodes related to each file. At first, the new implementation loads information about the ccp-change edges across the sub-OpGs for different files and restores a sub-OpG for a file with respect to a given slicing criteria. Then it picks up the needed files and restores their sub-OpGs on demand. This solution is likely to overcome the efficiency problem since a limited number of edit operations are related to each file in most cases and also cut/copy-and-paste actions across different files are not frequently executed.

5.3.3 Threats to Validity

Exp 1 and Exp 2 do not directly show practical benefits to actual programmers who have to understand how existing code has been written and modified. To obtain experimental results that show such benefits, we must design an experiment that simulates the programmers' tasks that are performed in actual software development and evolution. Unfortunately, this is obviously hard, and such an experiment is likely to include threats to the internal validity. Exp 1 and Exp 2 escape such threats and present only results in which operation history slicing achieves automatic and accurate reduction of the search space to understand the existing code. Several factors other than this reduction might bring practical benefits.

Moreover, Exp 1 and Exp 2 might include threats of the construct validity. We have no firm evidence that the five techniques, T1 ∼ T5, reflect typical programmers' tasks for understanding past code changes. Three metrics based on these techniques (size ratio, spreading ratio, and accuracy ratio) may not reflect the reduction of search space.

Of course, threats to the external validity exist. The results in Exp 1 and Exp 2 were derived from only three small-sized applications that were written by university students. Different results would be obtained from experiments with the operation history of code written by professional programmers in realistic software development or evolution. Moreover, all the applications are related to game programs with less variation in characteristics. Experimental results might depend on their target domains or quality. To check whether the results in Exp 1 and Exp 2 can be generalized to describe any situation in software development and evolution, a large number of experiments with edit operations for various applications must be made.

6. Related Work

Typical file-based version control systems (VCSs), such as CVS [17], Subversion (SVN) [18], and Git [28], store changes as line-based textual differences between two versions of a file that contains code and they allow programmers to scan them. In these systems, the collection of the differences is considered the change history. Unfortunately, several studies pointed out that the code changes stored in such histories are problematic because of their incompletion and imprecision [6], [11], [29], [30]. If a programmer changes the same code multiple times, a subsequent code change might override an earlier one. The overridden change might not be present in the history. If multiple changes are overlapped on the same piece of code, the history cannot recover precise information about those changes. For example, the time ordering of the changes is unclear or a lump of code appearing at a certain moment is broken. Programmers are required to identify the changes that have been actually made to a particular piece of code.

Several approaches based on IDE monitoring support programmers' tasks in software evolution. Mylyn [31] constructs a task context by capturing interactions between a programmer and her programming tools and computes a weighted value that represents the degree-of-interest (DOI) for each task. It reduces the amount of information the IDE displays based on DOI values. Similarly, HeatMap [32] highlights entities related to code by coloring based on past navigation, modification, or deletion of entities. NavTracks [33] and Team Tracks [34] record past navigation events that are performed on IDEs to help future programmers explore code. Such navigation seems to be replays of recorded IDE operations.

With respect to the automatic recording of programmer operations and their visualization, Project Watcher [35] resembles our idea. Furthermore, several techniques using the animated visualization of software history have been proposed. Evolution Storyboard [36] consists of a sequence of animated panels that represent the past changes of the structural dependency between two software artifacts. YARN [37] generates architecture-level animations of the changing dependency between sub-systems.

These IDE-based approaches can help programmers understand the existing code and its evolution. However, they do not directly replay edit operations for code performed on editors in IDEs. All the recorded information is highly abstracted during its analysis or visualization. Our goal is to provide a mechanism and its tool that makes replaying past code changes more efficient. This agrees with the goal of several studies [8], [9], [13], [14], [38], [39], including the work mentioned in Sect. 2.1, all of which focus on replaying fine-grained code changes and help programmers understand the code and its evolution. Although the granularity of the code changes treated in these studies slightly differs from that of the code changes relayed in our OperationSliceReplayer, the concept of operation history slicing can be applied to existing replaying tools.

Regarding the change relation and its graph representation, a few challenges are closely related to our study. Alam et al. proposed the concept of a time dependence relation between two structural changes on code, which indicates that one change to a code entity follows (depends on) another change [40]. Their approach uses information at the entity-level changes (changes related to functions, function calls, variables) that are lifted from the line-level changes stored in the code repository and constructs a time dependence relation among those entity-level changes. The time dependence relation helps programmers or managers track accurately and timely the progress of a project. A change impact graph (CIG) [41] and a genealogy of changes [42] are based on a concept that closely resembles time dependence, although they deal with code entities at different levels of granularity. They all represent information on the temporal dependence among code changes. From this point of view, our operation history graph (OpG) can be considered a variant of the aforementioned graphs. A big difference is the unit of code change. An OpG represents a dependence relation between finer-grained and more accurate code changes, i.e., edit operations actually performed on code by programmers, created using offset-level mapping instead of entity-level or line-level mapping.

Obviously, the closest study to this paper is history slicing [43], [44], which extracts a set of the lines of code of interest from the whole history of lines of code. A history slice is computed by traversing the history graph. Each of the vertices represents a line of code in a revision of a file. Each edge links a vertex in one specific revision and another vertex in its succedent revision, which is assigned using traditional line mapping techniques (e.g., [45]). The concepts of history slicing and our operation history slicing are the same but their mechanisms are vastly different. As mentioned before, our operation history slicing uses an OpG created by offset-level mapping instead of a history graph created by line-level mapping. Furthermore, our mapping is applied to entities in a parseable snapshot of code without concern for its revisions. On the other hand, history slicing strongly involves the revisions of code. Consequently, the two types of history slices have a big difference in their properties. A history slice is simply a set of lines of code, whereas an operation history slice is a set of edit operations, each of which contains information on past code additions and/or deletions. In other words, our operation history slice is applied to code as an edit script that can then be replayed. To demonstrate the possibility of replaying, we developed a running implementation.

History reordering [46] also deals with edit operations recorded by OperationRecorder, in common with our operation history slicing. However, the goal of history reordering is not to reduce the amount of information on the history in its understanding, but to restructure the mixed changes to make them easier to reuse, revert, and understand.

7. Conclusion

For understanding its evolution, replaying past edit operations for code is useful but is often time-consuming. This paper presented a mechanism of operation history slicing that can automatically eliminate the nonessential skipping of edit operations for class members of no interest. This mechanism employs an operation history graph that represents the impact of edit operations on snapshots of code.

The development of OperationSliceReplayer continues. Two immediate issues remain for its enhancement. It currently treats inner or anonymous classes as part of class members (methods to be exact) including these classes. To separate such classes from their respective outer class members, nesting the offset ranges of the class members and the classes enclosed by them will be considered.

Our future work will expand the definition of an OpG, which currently contains only inter-snapshot dependence relations related to code changes. However, conventional (intra-snapshot) relations obtained from cross-referencing information (program dependency or method call) within a program might be valuable. For example, replaying the construction of one method would likely involve replaying the construction of another method called by the method. The concept of time dependence relation [40] is also worth introducing into our mechanism.

Acknowledgments

This work was partially sponsored by the Grant-in-Aid for Scientific Research (24500050, 26730042, 15H02685, 15K15970) from the Japan Society for the Promotion of Science (JSPS).

References

[1] A. von Mayrhauser and A.M. Vans, "Program comprehension during software maintenance and evolution," Computer, vol.28, no.8, pp.44–55, 1995.
[2] T.D. LaToza and B.A. Myers, "Hard-to-answer questions about code," Proc. PLATEAU '10, pp.8:1–8:6, 2010.
[3] T.D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: A study of developer work habits," Proc. ICSE '06, pp.492–501, 2006.
[4] A.J. Ko, R. DeLine, and G. Venolia, "Information needs in collocated software development teams," Proc. ICSE '07, pp.344–353, 2007.
[5] A.W.J. Bradley and G.C. Murphy, "Supporting software history exploration," Proc. MSR '11, pp.193–202, 2011.
[6] R. Robbes and M. Lanza, "A change-based approach to software evolution," Proc. ERCIM Working Group on Software Evolution, Electronic Notes in Theoretical Computer Science (ENTCS), vol.166, pp.93–109, Elsevier, Jan. 2007.
[7] P. Ebraert, J. Vallejos, P. Costanza, E.V. Paesschen, and T. D'Hondt, "Change-oriented software engineering," Proc. ICDL '07, pp.3–24, 2007.
[8] R. Robbes and M. Lanza, "SpyWare: A change-aware development toolset," Proc. ICSE '08, pp.847–850, 2008.
[9] L. Hattori and M. Lanza, "Syde: A tool for collaborative software development," Proc. ICSE '10, pp.235–238, 2010.
[10] Y. Yoon and B.A. Myers, "Capturing and analyzing low-level events from the code editor," Proc. PLATEAU '11, pp.25–30, 2011.
[11] S. Negara, M. Vakilian, N. Chen, R.E. Johnson, and D. Dig, "Is it dangerous to use version control histories to study source code evolution?," Proc. ECOOP '12, Lecture Notes in Computer Science, vol.7313, pp.79–103, Springer Berlin Heidelberg, 2012.
[12] T. Omori and K. Maruyama, "A change-aware development environment by recording editing operations of source code," Proc. MSR '08, pp.31–34, 2008.
[13] R. Robbes, D. Pollet, and M. Lanza, "Replaying IDE interactions to evaluate and improve change prediction approaches," Proc. MSR '10, pp.161–170, 2010.
[14] L. Hattori, M. Lungu, and M. Lanza, "Replaying past changes in multi-developer projects," Proc. IWPSE-EVOL '10, pp.13–22, 2010.
[15] T. Omori and K. Maruyama, "An editing-operation replayer with highlights supporting investigation of program modifications," Proc. IWPSE-EVOL '11, pp.101–105, 2011.
[16] Y. Yoon, B.A. Myers, and S. Koo, "Visualization of fine-grained code change history," Proc. VL/HCC '13, pp.119–126, 2013.
[17] "CVS — Concurrent versions system," http://www.nongnu.org/cvs/
[18] "Apache subversion," http://subversion.apache.org/
[19] L. Hattori, M. D'Ambros, M. Lanza, and M. Lungu, "Software evolution comprehension: Replay to the rescue," Proc. ICPC '11, pp.161–170, 2011.
[20] C. Parnin and R. DeLine, "Evaluating cues for resuming interrupted programming tasks," Proc. CHI '10, pp.93–102, 2010.
[21] K. Maruyama, E. Kitsu, T. Omori, and S. Hayashi, "Slicing and replaying code change history," Proc. ASE '12, pp.246–249, 2012.
[22] M. Weiser, "Program slicing," IEEE Trans. Softw. Eng., vol.SE-10, no.4, pp.352–357, 1984.
[23] M. Harman, S. Danicic, and Y. Sivagurunathan, "Program comprehension assisted by slicing and transformation," UK Program Comprehension Workshop, Durham University, 1995.
[24] M.A. Storey, "Theories, methods and tools in program comprehension: Past, present and future," Proc. IWPC '05, pp.181–191, 2005.
[25] Eclipse.org, "Eclipse," http://www.eclipse.org/
[26] Y. Yoon and B.A. Myers, "An exploratory study of backtracking strategies used by developers," Proc. CHASE '12, pp.138–144, 2012.
[27] Y. Yoon and B.A. Myers, "Supporting selective undo in a code editor," Proc. ICSE '15, pp.223–233, 2015.
[28] "Git," http://git-scm.com/
[29] E. Kitsu, T. Omori, and K. Maruyama, "Detecting program changes

from edit history of source code," Proc. APSEC '13, pp.299–306, 2013.

[30] K. Herzig and A. Zeller, "The impact of tangled code changes," Proc. MSR '13, pp.121–130, 2013.

[31] M. Kersten and G.C. Murphy, "Using task context to improve programmer productivity," Proc. FSE '06, pp.1–11, 2006.

[32] D. Röthlisberger, O. Nierstrasz, S. Ducasse, D. Pollet, and R. Robbes, "Supporting task-oriented navigation in IDEs with configurable HeatMaps," Proc. ICPC '09, pp.253–257, 2009.

[33] J. Singer, R. Elves, and M.A. Storey, "NavTracks: Supporting navigation in software maintenance," Proc. ICSM '05, pp.325–334, 2005.

[34] R. DeLine, M. Czerwinski, and G. Robertson, "Easing program comprehension by sharing navigation data," Proc. VL/HCC '05, pp.241–248, 2005.

[35] K.A. Schneider, C. Gutwin, R. Penner, and D. Paquette, "Mining a software developer's local interaction history," Proc. MSR '04, pp.106–110, 2004.

[36] D. Beyer and A.E. Hassan, "Animated visualization of software history using evolution storyboards," Proc. WCRE '06, pp.199–210, 2006.

[37] A. Hindle, Z.M. Jiang, W. Koleilat, M.W. Godfrey, and R.C. Holt, "YARN: Animating software evolution," Proc. VISSOFT '07, pp.129–136, 2007.

[38] R. Robbes, "Mining a change-based software repository," Proc. MSR '07, pp.15–22, 2007.

[39] R. Robbes and M. Lanza, "Characterizing and understanding development sessions," Proc. ICPC '07, pp.155–166, 2007.

[40] O. Alam, B. Adams, and A.E. Hassan, "Measuring the progress of projects using the time dependence of code changes," Proc. ICSM '09, pp.329–338, 2009.

[41] D.M. German, G. Robles, and A.E. Hassan, "Change impact graphs: Determining the impact of prior code changes," Proc. SCAM '08, pp.184–193, 2008.

[42] I.I. Brudaru and A. Zeller, "What is the long-term impact of changes?," Proc. RSSE '08, pp.30–32, 2008.

[43] F. Servant and J.A. Jones, "History slicing," Proc. ASE '11, pp.452–455, 2011.

[44] F. Servant and J.A. Jones, "History slicing: Assisting code-evolution tasks," Proc. FSE '12, pp.1–11, 2012.

[45] G. Canfora, L. Cerulo, and M.D. Penta, "Identifying changed source code lines from version repositories," Proc. MSR '07, pp.14–21, 2007.

[46] S. Hayashi and M. Saeki, "Recording finer-grained software evolution with IDE: An annotation-based approach," Proc. IWPSE-EVOL '10, pp.8–12, 2010.

Katsuhisa Maruyama received his B.E. and M.E. degrees in electrical engineering and Ph.D. in information science from Waseda University, Japan, in 1991, 1993, and 1999, respectively. He is a professor of the Department of Computer Science, Ritsumeikan University. He has worked for NTT (Nippon Telegraph and Telephone Corporation) and NTT Communications Corporation before he joined Ritsumeikan University. He was a visiting researcher at Institute for Software Research (ISR) of University of California, Irvine (UCI). His research interests include software refactoring, program analysis, software reuse, object-oriented design and programming, and software development environments. He is a member of the IEEE Computer Society, ACM, IPSJ, and JSSST.

Takayuki Omori is a lecturer at the Department of Computer Science, Ritsumeikan University. He obtained his Ph.D. in Engineering from Ritsumeikan University in 2008. From September 2013 to March 2014, he was a visiting assistant professor at the University of British Columbia. His current research interests include software evolution, and fine-grained code changes in software development.

Shinpei Hayashi received his B.E. degree in information engineering from Hokkaido University in 2004. He also received his M.E. and Ph.D. degrees in computer science from Tokyo Institute of Technology in 2006 and 2008, respectively. He is currently an assistant professor of computer science at Tokyo Institute of Technology. His research interests include software changes and software development environments.