Slicing and Replaying Code Change History

Katsuhisa Maruyama[1], Eijiro Kitsu[2], Takayuki Omori[1], Shinpei Hayashi[3]
[1] Department of Computer Science, Ritsumeikan University, Japan
[2] Graduate School of Science and Engineering, Ritsumeikan University, Japan
[3] Department of Computer Science, Tokyo Institute of Technology, Japan
maru@cs.ritsumei.ac.jp, {kitsu, takayuki}@fse.cs.ritsumei.ac.jp,
hayashi@se.cs.titech.ac.jp

ABSTRACT

Change-aware development environments have recently become feasible and reasonable. These environments can automatically record fine-grained code changes on a program and allow programmers to replay the recorded changes in chronological order. However, they do not always need to replay all the code changes to investigate how a particular entity of the program has been changed. Therefore, they often skip several code changes of no interest. This skipping action is an obstacle that makes many programmers hesitate in using existing replaying tools. This paper proposes a slicing mechanism that can extract only code changes necessary to construct a particular class member of a Java program from the whole history of past code changes. In this mechanism, fine-grained code changes are represented by edit operations recorded on source code of a program. The paper also presents a running tool that implements the proposed slicing and replays its resulting slices. With this tool, programmers can avoid replaying edit operations nonessential to the construction of class members they want to understand.

Categories and Subject Descriptors

D.2.6 [**Programming Environments**]: Integrated environments; D.2.7 [**Distribution, Maintenance, and Enhancement**]: Version control; D.2.3 [**Coding Tools and Techniques**]: Program editors

General Terms

Algorithms, Human Factors

Keywords

Software maintenance and evolution, Program comprehension, Program slicing, Code change, Integrated development environments

1. INTRODUCTION

In software maintenance, many programmers (mainly maintainers) must often understand existing source code that someone else has written or modified before [12]. To make such understanding easier, change-based support has recently become popular [3, 9]. In general, programmers do not only concentrate on examining source code but would look at its time-series data such as a chronological sequence of its snapshots. This enables them to obtain knowledge of how the source code has been ever changed. For example, SpyWare [10], Syde [7], and OperationRecorder [8] are embedded into modern integrated development environments (IDEs) and capture all fine-grained code changes performed on the editors provided by their respective IDEs. In addition, these recording tools collaborate with tools visualizing, filtering, and/or replaying recorded code changes.

Using these tools helps programmers keep track of fine-grained code changes individually stored in the repository. In particular, a replaying tool can reenact past programming scenes in front of programmers' faces. This makes them to image what other programmers have gone through in the past. Moreover, the history of past code changes provides hints at programmers' understanding how to revert undesired code changes based on their decisions made in the past. For example, a controlled experiment conducted by Hattori et al. [6] demonstrated that chronologically replaying of fine-grained code changes outperforms existing commit-based versioning systems on helping programmers find answers to questions related to software evolution.

Although chronologically replaying of fine-grained code changes of a program is useful for understanding its evolution, we emphasize the possibility of improvement in assistance for replaying. In general, replaying is a time-consuming task. If huge amount of code changes were recorded, it takes a long time to replay every change. In most cases, programmers do not need to investigate the whole evolution of source code. They incrementally obtain knowledge on past code changes by partially replaying the code changes depending on their interests. To encourage programmers to exploit existing replaying tools, automatic extraction of code changes to be replayed is required. This helps them efficiently understand the evolution of a particular part of source code. Here, careful readers might think that edit operations not related to a program entity of interest can be filtered out by checking its name. Unfortunately, simple filtering provided by the conventional tools does not address fine-grained tracking of code changes resulting from the renaming, splitting, or merging of program entities (e.g., methods of a Java program), or the moving or copying of part of their bodies through a cut-paste or copy-paste action.

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 [13]. Program slicing is used to extract from code of a program a set of statements that may affect (the calculation of) the value of a variable of interest at a specified program point. Here, the princi-

ASE'12, September 3–7, 2012, Essen, Germany

pal concern is simplification when slicing to assist program comprehension [5] although much of application of slicing has been provided. Our idea exploits this simplification power of slicing in replaying past code changes.

In the mechanism, fine-grained code changes are represented by edit operations on source code of a Java program. Moreover, a special graph, called an *edit operation graph* (OpG hereafter), is introduced. It links class members (methods and fields) within snapshots of program code via edit operations performed on their respective snapshots. By traversing vertices and edges of an OpG, the mechanism can extract only edit operations necessary to construct (create, remove, and modify) a class member of interest from the history consisting of all recorded edit operations. Such extraction process and a collection of the extracted edit operations are called an *operation history slicing* and an *operation history slice*, respectively. This paper also presents a running tool, OperationSliceReplayer, which implements operation history slicing and replays its resulting slices.

The important feature of the proposed slicing is that the contents restored by replaying only edit operations included in an operation slice for a target class member on an arbitrary snapshot of source code are likely to be always the same as the original contents of the target class member. Due to this feature, if a programmer wants to understand a particular class member in his/her task, only edit operations included in its operation history slice would be replayed. In other words, programmers can avoid any nonessential action skipping edit operations of class members they have no interest in. Consequently, OperationSliceReplayer has the potential to make their program understanding tasks more efficient.

2. OPERATION HISTORY SLICING

It is obviously assumed that all edit operations with respect to manual and automatic code changes performed on the editor are completely collected. We adopt OperationRecorder [8] as a recording tool, which can automatically record edit operations that affect source code in Eclipse's Java editor. The operations include manual typing (insertion, deletion, and replacement of a text), editing via a clipboard (copy, cut, and paste of a text), undo/redo actions, and code changes by automatic transformation (code completion, quick fix, formatting, and refactoring).[1]

This section first describes a graph that represents relationships among class members of source code and recorded edit operations, and then defines operation history slicing using this graph.

2.1 Edit Operation Graph

To collect all edit operations constructing a particular class member of source code without omission, it is necessary to formulate the relationships between edit operations and code fragments affected by them. An *edit operation 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 that an edit operation was never performed on. The subscript number is incremented by one once each edit operation is applied. Here, p_i denotes the i-th edit operation, and S_i indicates the snapshot of the source code generated immediately after p_i was applied to its precedent snapshot (S_{i-1}). In other words, S_i can be obtained after all edit operations between p_1 and p_i are chronologically applied to S_0. Moreover, a snapshot consisting of only code fragments with no syntax error is called a parseable one.

[1]The current version of OperationRecorder excludes recording of actions related to file renaming and removing.

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

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

The i is the index number represents a subscript of a snapshot or an edit operation. The z is the index number represents a subscript of the latest snapshot S_z and one of the latest edit operation p_z. The $v.m$ denotes a vertex corresponding to the class member m, and the $v.p$ denotes a vertex corresponding to the edit operation p.

Next consider edges which link between 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, there is no snapshot S_k between S_i and S_j under $j = i+1$ or every snapshot S_k that satisfies $i < k < j$ is not parseable. Edges of an OpG are divided into the following four types.

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

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

(c) Consider a situation that the contents of a class member m within S_i remains in a class member m' within S_j without any change. If m has no forward-change edge, m' has no backward-change edge, and the full names of m and m' are the same, $v.m$ ($\in V$) and $v.m'$ ($\in V$) are linked by a *no-change* (unchanged) edge $v.m \rightarrow_n v.m'$ in the OpG G. The full name is a unique identifiable name constructed by concatenating the fully qualified name of a class, a special character "#", and the signature of a method or the name of a field. For example, "`fqn#sig`" and "`fqn#vn`" are the full names for a method with the signature "`sig`" and a field with the variable name "`vn`" of a class "`fqn`", respectively.

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

E is a set of all 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 $G = (V, E)$.

Figure 1 depicts part of an OpG representing the construction of a sample Java program with 26 lines of code. The OpG contains all

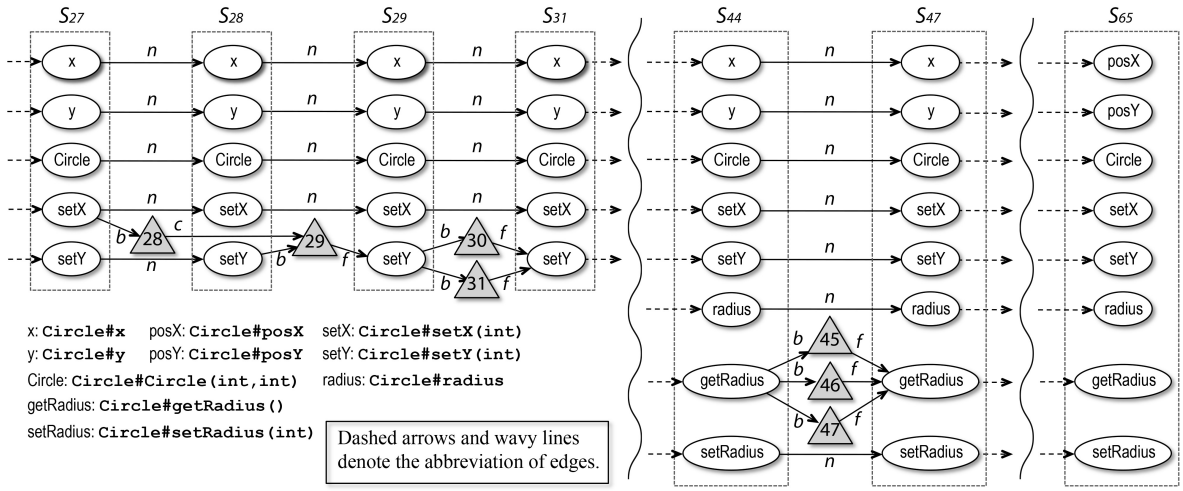

Figure 1: Part of an operation history graph (OpG). The whole OpG consists of 239 vertices and 243 edges.

65 edit operations performed during the construction and all class members appearing in its snapshots. Each triangle indicates a vertex corresponding to an edit operation. For example, p_{30}, p_{31}, p_{45}, p_{46}, and p_{47} are normal edit operations. Moreover, p_{28} is a copy operation and p_{29} is a paste operation. Each oval indicates a vertex corresponding to a class member. A dotted rectangle indicates a snapshot that a class member belongs to. Arrows marked with the labels *b*, *f*, *n*, and *c* indicate the four types of edges: a backward-change edge, a forward-change edge, a no-change edge, and a ccp-change edge, respectively.

To detect backward-change and forward-change edges between an edit operation and a class member of an OpG, their offset values are compared. The offset value is stored into information on each edit operation, which locates the starting point of its inserted, deleted, or copied text. In this detection, several offset values would be adjusted according as the length of the inserted or deleted text of their neighboring edit-operations. For example, for the OpG shown in Figure 1, S_{44} and S_{47} are adjacent parseable snapshots (i.e., S_{45} and S_{46} are not parseable). Thus, the offset values of p_{45}, p_{46}, and p_{47} between S_{44} and S_{47} are candidates to be adjusted. In fact, the offset value of p_{46} was adjusted. After the detection of all backward-change and forward-change edges, no-change and ccp-change edges will be detected. These edges can be easily detected. For example, p_{28} and p_{29} can be linked via a ccp-change edge.

2.2 Operation History Slice

By traversing vertices and edges of an OpG, the proposed mechanism can extract edit operations necessary to construct a class member of interest from the history consisting of all edit operations. Here, G_S is an OpG for source code S. Let $R(G_S, v.m)$ be a set of vertices of G_S that reach a vertex $v.m$ corresponding to a class member m within a 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 vertices of G_S. The relation $\rightarrow^*$ means the reflexive and transitive closure of the relation $\rightarrow$, which indicates one of the four types of edges ($\rightarrow_b$, $\rightarrow_f$, $\rightarrow_n$, or $\rightarrow_c$) of G_S.

Here, there exists sometime an edit operation having no forward-change edge in the OpG. For example, this occurs when an edit operation deletes or cuts the whole contents of a method or a field. Since such 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 $Op(G_S, v.m)$ is defined as follows:

$$Op(G_S, v.m) = \{\, u \in V_p(G_S) \mid u \in R'(G_S, v.m) \,\}.$$

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

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

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

For every recorded edit operation, p_1 is the first (earliest) one and p_z are the last (latest) one. The above sequence is drawn up in their 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.[2] In case that there exists only one edit operation for S, $Q(S) = \langle p_1 \rangle$.

An operation slice $Sq(S, m)$ is a minimal sub-sequence of $Q(S)$ that satisfies the following condition:

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

$Q_1 \sqsubseteq Q_2$ 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. The m is a slicing criterion that denotes a class member of interest within a snapshot S.

The snapshot S_{65} (the final code) for the OpG shown in Figure 1 has eight class members (three fields and five methods). For a method `setX()` within S_{65}, its operation history slice is as follows:

$$Sq(G, \texttt{setX@}S_{65}) = \langle 13, 14, 15, 16, 17, 18, 19, 20, 28, 60, 61 \rangle.$$

The slice contains 11 edit operations. Thus, its ratio to the total number of the recorded edit operations is 16.9% (= 11/65).

Figure 2 shows the screenshot of the OperationSliceReplayer perspective in Eclipse. OperationSliceReplayer chronologically restores

[2] If two edit operations are performed at the same time, their chronological order is not uniquely determined. Although this happens, OperationRecorder suitably orders and records them.

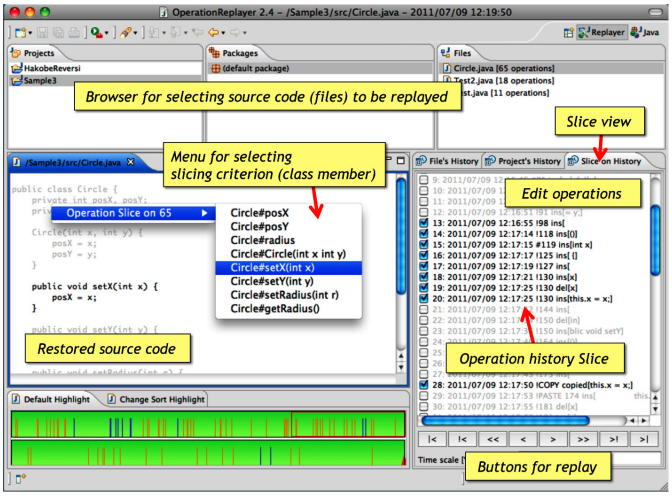

Figure 2: OperationSliceReplayer perspective in Eclipse.

the contents of a source file of interest and presents an animated movie tracking its changes, on the left view of the perspective. When a programmer wants to see the growth of a particular class member, he/she can activate a slicing menu presenting all class members existing in currently restored source code. If it is not parseable, OperationSliceReplayer tries to find an immediately precedent parseable snapshot and presents its class members. Recorded edit operations are listed on the right view. In this view, edit operations included in the operation history slice are displayed in the black, and ones not included in the slice are displayed in the grey. He/She can replay one-by-one (plus rewind and fast-forward) the whole history of the recorded edit operations, and also skip unnecessary edit operations by pushing each of the replay buttons.

3. RELATED WORK

Obviously, our goal is in agreement with that of the study by Hattori et al. [6, 7], and the concept of operation history slicing can be applied to their replaying tool.

Regarding the change relation and its graph representation, a few challenges are closely related to our study. Alam et al. [1] proposed the concept of a time dependence relation between two structural changes on source code, which indicates that one change to a source code entity follows (depends on) another change. In addition, a change impact graph (CIG) [4] or a genealogy of changes [2] is based on the concept almost the same as the time dependence. They all represent information on the temporal dependence between code changes. From this point of view, our OpG can be considered a variant of the aforementioned graphs. A big difference is what is the unit of code change. The OpG represents a dependence relation between finer-grained and more accurate code changes, which is built by using the offset-level mapping instead of the entity-level mapping.

History slicing [11] is the closest study. It extracts a set of lines of code of interest from its whole history. The concept of the history slicing and that of our operation history slicing are the same but their mechanisms are vastly different. Our operation history slicing uses the OpG built by offset-level mapping instead of the history graph built by line-level mapping. As a result, the history slice is simply a set of lines of code, whereas the operation history slice is a set of edit operations each of which contains information on past code addition and/or deletion. In other words, our slice is applied to code as an edit script and then can be replayed.

4. CONCLUSION

Replaying past edit operations for source code is useful for understanding its growth but is often a time-consuming task. This paper has presented a mechanism of operation history slicing that can automatically eliminate nonessential skipping of edit operations for class members of no interest. We have still no firm evidence for the benefits of its use during program comprehension. To check whether operation history slicing can reduce program comprehension effort, we must make a large number of comparative experiments with and without introducing this slice.

The development of OperationSliceReplayer is continuing. Two immediate issues mainly remain in its enhancement. It currently treats inner classes as a part of class members (methods to be exact) including these classes. To separate such classes from their respective outer class members, nesting offset ranges of class members and classes enclosed by them is considered. Moreover, OperationSliceReplayer should support renaming of classes. The current implementation cannot detect non-change edges in this situation.

5. ACKNOWLEDGMENTS

This work was partially sponsored by the Grant-in-Aid for Scientific Research (23700030, 24500050, 24700034) from the Japan Society for the Promotion of Science (JSPS).

6. REFERENCES

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

[2] I. I. Brudaru and A. Zeller. What is the long-term impact of changes? In *Proc. RSSE'08*, pages 30–32, 2008.

[3] P. Ebraert, J. Vallejos, P. Costanza, E. V. Paesschen, and T. D'Hondt. Change-oriented software engineering. In *Proc. ICDL'07*, pages 3–24, 2007.

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

[5] M. Harman, S. Danicic, and Y. Sivagurunathan. Program comprehension assisted by slicing and transformation. In *UK Program Comprehension Workshop*. Durham Univ., 1995.

[6] L. Hattori, M. D'Ambros, M. Lanza, and M. Lungu. Software evolution comprehension: Replay to the rescue. In *Proc. ICPC'11*, pages 161–170, 2011.

[7] L. Hattori and M. Lanza. Syde: A tool for collaborative software development. In *Proc. ICSE'10*, pages 235–238, 2010.

[8] T. Omori and K. Maruyama. A change-aware development environment by recording editing operations of source code. In *Proc. MSR'08*, pages 31–34, 2008.

[9] R. Robbes and M. Lanza. A change-based approach to software evolution. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 166:93–109, January 2007.

[10] R. Robbes and M. Lanza. SpyWare: A change-aware development toolset. In *Proc. ICSE'08*, pages 847–850, 2008.

[11] F. Servant and J. A. Jones. History slicing. In *Proc. ASE'11*, pages 452–455, 2011.

[12] A. von Mayrhauser and A. M. Vans. Program comprehension during software maintenance and evolution. *IEEE Computer*, 28:44–55, 1995.

[13] M. Weiser. Program slicing. *IEEE TSE*, 10(4):352–357, 1984.