Published version: Ryosuke Funaki, Shinpei Hayashi, Motoshi Saeki, "The Impact of Systematic Edits in History Slicing,", in Proceedings of the 16th International Conference on Mining Software Repositories, pp. 555–559, 2019, https://doi.org/10.1109/MSR.2019.00083

The Impact of Systematic Edits in History Slicing

Ryosuke Funaki, Shinpei Hayashi, and Motoshi Saeki
Tokyo Institute of Technology, Tokyo 152–8550, Japan
Email: {rfunaki,hayashi,saeki}@se.cs.titech.ac.jp

***Abstract*—While extracting a subset of a commit history, specifying the necessary portion is a time-consuming task for developers. Several commit-based history slicing techniques have been proposed to identify dependencies between commits and to extract a related set of commits using a specific commit as a slicing criterion. However, the resulting subset of commits become large if commits for systematic edits whose changes do not depend on each other exist. We empirically investigated the impact of systematic edits on history slicing. In this study, commits in which systematic edits were detected are split between each file so that unnecessary dependencies between commits are eliminated. In several histories of open source systems, the size of history slices was reduced by 13.3–57.2% on average after splitting the commits for systematic edits.**

***Keywords*-version control; history slicing; systematic edits;**

I. Introduction

Version control systems such as Git [1] are widely used for software maintenance. The use of version control systems enables developers to easily manage product releases and merge the implementation of new features by pull requests.

Developers are sometimes required to extract a subset of their history from the whole [2], e.g., while *backporting* changes from one branch to another or applying *branch refactoring* to decompose a branch into multiple subsets. [2]. In backporting, major changes such as adding features or fixing bugs are committed to the main branch, and only the necessary subset is cherry-picked and applied to maintenance branches. In branch refactoring, in situations where unrelated changes are committed together in a single branch, developers untangle the commits in the branch and split them into multiple ones to improve understandability and portability. This refactoring activity is done while extracting only the necessary changes to create pull requests.

The extraction of a subset of commits may fail [3]. Because changes in commits depend on each other, extracting a subset from a history must take such dependencies into consideration, i.e., if a commit c is included in the subset of changes, another commit c' on which c depends must also be included. Because the manual identification of these dependencies is a time-consuming task for developers, an automated mechanism is required.

Applications of a slicing to a history structure (hereafter, *history slicing*) [2], [4]–[6] are useful in automating the extraction of a near-sufficient set of required changes. These approaches were inspired by the concept of program slicing [7], which extracts a set of statements that might affect the value of a variable of interest at a specified program point from the code of a program, to be used as a slicing criterion. This extracted code is called a *slice* and is computed by a graph reachability algorithm for the program dependence graph of a target program. In history slicing, a dependence graph of change elements of a history is prepared, and a subset is extracted as a *history slice*. Change elements of different types, e.g., lines [4], [5], edit operations [6], or commits [2], can be considered as the application target of the history slicing.

Commits of larger sizes with a wider spread of dependencies can lead to a substantial increase in the size of history slices. In particular, commits consisting of independent sets of code lead to unnecessary dependencies among changes. Examples of such changes are non-essential changes [8], tangled changes [9]–[12], and impure refactoring changes [13], [14]. In addition, *systematic edits* [15], [16], i.e., similar edits to multiple locations of source code, often happen in a history and are troublesome to history slicing. Because history slices are required to be as small as possible, avoiding unnecessary magnification is important. However, the impact of systematic edits in history slicing has not yet been investigated.

The purpose of this paper is to investigate the impact of systematic edits on commit-based history slicing. In particular, we will answer the research question: *How much do systematic-edit-based commits impact the size of commit-based history slices?* (the first contribution). Based on the empirical study of two open-source systems in Apache Commons ecosystem, we found that a reasonable number of unnecessary changes were included. To conduct this study, we enhanced commit-based history slicing to make a traditional approach systematic-edits-aware (the second contribution).

The rest of this paper is organized as follows. Section II explains our motivation using a concrete example. Section III explains our enhancement of commit-based history slicing. Section IV shows an empirical study to answer the research question. Section V concludes this paper.

II. Motivation

Commits consisting of unrelated changes lead to the enlargement of history slices. Consider the example shown in Fig. 1, which illustrates a history slice obtained from Apache Commons Collections. The commit of `51186c1`[1] is used as the slicing criterion for the commit history of versions 4.1–4.2, which consists of 111 commits. This commit depends on another commit `059c468`[2], which adds the qualifier `final` to many fields and variable declarations over 82 files. Among

[1]https://github.com/apache/commons-collections/commit/51186c1
[2]https://github.com/apache/commons-collections/commit/059c468

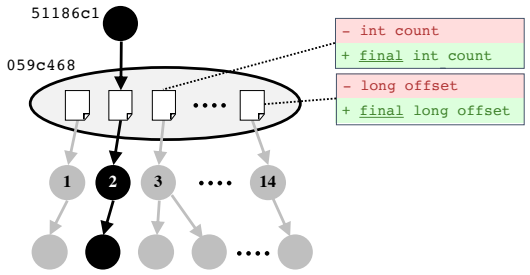


Fig. 1. Example of history slicing for Commit `059c468`.

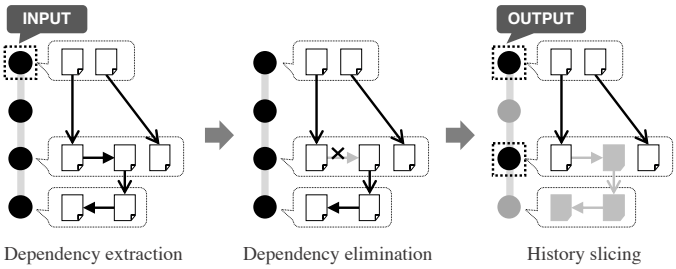


Fig. 2. Overview of our history slicing.

these, the changed lines in 14 files overwrote other changes in other commits, which implies textual dependencies on 14 changed files. This means that it is necessary to capture 14 additional commits to capture `059c468`, which leads to the enlargement of the resulting history slice. This issue is owing to the commit level granularity of sliced elements in history slicing, i.e., all changes of `059c468` were included. In fact, because each addition of the `final` qualifier does not affect the others, they are independently applicable. If file changes in the commit can be handled independently on a changed file level, the total number of changes included in the resulting history slice will be reduced, and the resulting slice will become smaller.

III. Systematic-Edit-Aware History Slicing

A. Overview

An overview of our technique of systematic-edit-aware commit-based history slicing is shown in Fig. 2. Its input is the target commit history and a specific commit for the slicing criteria. Its output is a history slice consisting of a subset of changed files in the given history. This process consists of three steps: *dependency extraction*, *dependency elimination*, and *history slicing*. During dependency extraction, the dependencies between all change elements in the target commit history are extracted. Dependency elimination detects the commits consisting of systematic edits and eliminates dependencies of a specific type that are regarded as unnecessary. In history slicing, we apply history slicing using the change elements in the given commit as slicing criteria to obtain a set of related change elements.

B. Dependency Extraction

In our approach, the basic elements in history slicing are *files* modified in commits rather than *commits* that are used in existing techniques [2]. We aim at a finer-grained extraction of history slices because we think that the changes in some commits are inappropriate to be extracted at once.

We call a pair $e = \langle c, f \rangle \in E$ of a commit $c \in C$ and a file modified in the commit $f \in F$ *change element*. The dependencies between change elements are expressed as a binary relation $\rightarrow \subseteq E \times E$. We define the dependencies as a union of three different types of dependencies.

- A *textual dependency* between change elements $e \rightarrow_h e'$ exists if the modified line range, which includes its context lines before and after modification, in e overlaps with that in a past change e'. More specifically, the dependency can be expressed as

$$\rightarrow_h = \{ \langle e, e' \rangle \mid range(e) \cap range(e') \neq \emptyset \}$$

  where $range(e)$ is the modified line range of e.
- A *build dependency* between change elements $e \rightarrow_b e'$ exists if e' are essential for the successful build of the final snapshot of any sub-history including e. More specifically, the dependency can be expressed as

$$\rightarrow_b = \{ \langle e, e' \rangle \mid \forall E' \subseteq E \bullet e \in E' \wedge \checkmark_{build}(E') \Longrightarrow e' \in E' \}$$

  where $\checkmark_{build}(E')$ expresses that the build succeeds with the final snapshot of the sub-history E'.
- A *commit dependency* between change elements $e \rightarrow_c e'$ exists if they belongs to the same commit:

$$\rightarrow_c = \{ \langle \langle c, f \rangle, \langle c', f' \rangle \rangle \mid c = c' \}.$$

C. Dependency Elimination

Among the commit dependencies obtained in the previous step, those eliminable are identified. If there is no semantic relationship between change elements that occurred at the same commit, the commit dependencies between them can be eliminated.

Systematic edits [15], [16] are considered a representative example of changes that do not depend on each other but are committed at once. Systematic edits are typically done automatically by tools such as those provided by IDEs. The resulting changes in a certain file are considered to not affect the other files of the same systematic editing. For example, the set of changes adding the qualifier `final` shown in Fig. 1 is a typical instance set of systematic edits.

Molderez et al. [17] proposed a technique to detect systematic edits by abstracting edit scripts of source code changes obtained from differencing abstract syntax trees (ASTs) of Java source code and grouping them by applying frequent itemset mining. In this technique, an edit script is expressed as a tuple of $\langle changeType, structuralSubject, location \rangle$ where *changeType* is the type of the change, *structuralSubject* is an abstracted representation of the AST node to be changed, and *location* is the absolute path of the node. The mining method is applied to a set of obtained edit scripts, and the mined frequent patterns are regarded as systematic edits.

TABLE I
TARGET SYSTEMS

Project	History (Versions)	First commit	Last commit	# Commits	# Systematic
Commons Collections	$CC_{4.0}$ (4.0–4.1)	`5950eba` (2013–11–21)	`a7cbb44` (2015–11–26)	211	13
	$CC_{4.1}$ (4.1–4.2)	`1ce3b3e` (2015–11–28)	`483cbbb` (2018–07–08)	170	35
	CC_{all} (1.0–4.2)	`3f06f58` (2001–07–15)	`483cbbb` (2018–07–08)	2,971	330
Commons Net	$CN_{3.3}$ (3.3–3.4)	`dc0da97` (2013–06–08)	`6f97833` (2015–11–19)	270	31
	CN_{all} (1.0–3.6)	`564a20c` (2003–02–19)	`e207b99` (2017–02–11)	1,954	146

We detect systematic edits using Molderez et al.'s technique with an extension. Our aim is to precisely find *splittable* commits to decrease the dependencies among change elements. To achieve the accurate detection of such commits, we only focus on commits that can be represented as a single set of systematic edits, i.e., all the changes in the target commit belong to the same instance set of systematic edits. For this purpose, we do not apply the frequent pattern mining. We apply an abstraction to edit scripts and check whether all the scripts for all the changed methods follow the same manner. More specifically, we regard a set of edit scripts at the commit c as systematic if the following condition holds:

$$\forall m_1, m_2 \in M_c \bullet chg_c(m_1) = chg_c(m_2)$$

where M_c is a set of changed methods at c and $chg_c(m)$ is the set of abstracted edit scripts for the changes in m at c.

For the abstraction to be applied to edit scripts, we slightly changed the approach of Molderez et al.:

- We omit the information on the absolute path stored in *location* because we want to regard changes of different types at different locations to be similar. See again the example shown in Fig. 1; the addition of the qualifier `final` occurred at identifiers of different types such as local variables or formal parameters of methods. This rule can regard them as the same.
- The pair of code fragments before and after the code change is used as the information representing the target AST node stored in *structuralSubject*. Molderez et al.'s technique used only either of them: code fragments after change for addition and modification; those before change for deletion. This was insufficient to check whether all the changes applied in the same commit satisfied the above conditions.

We regarded the changes of a commit as systematic if all the abstracted edit scripts of them follow the conditions shown above. In addition, commits consisting of only white-spaces or comment changes without any syntax tree differences are also systematic of a special type.

If commits consisting of systematic edits are found, we collect the commit dependencies of these commits $\tilde{\rightarrow}_c$. These commit dependencies are eliminated from the set of all dependencies: $\rightarrow'_c = \rightarrow_c \setminus \tilde{\rightarrow}_c$. The history slices are then calculated using the updated dependencies.

D. History Slicing

For the change elements contained in the input commits E_{init}, this step recursively traces the dependencies obtained in the previous steps $\rightarrow = \rightarrow_h \cup \rightarrow_b \cup \rightarrow'_c$ and retrieves all the change elements that are traceable by the dependencies as a history slice:

$$E^* = \bigcup_{e \in E_{init}} \{ e' \mid e \rightarrow^* e' \}$$

where $\rightarrow^*$ is a transitive closure of the dependencies $\rightarrow$.

The list of commits is then extracted based on the obtained slice. All the non-systematic-edit-based commits containing at least one change element in E^* are cherry-picked and included in the output. For systematic-edit-based commits, they are reconfigured so only the necessary change elements are included and are committed to the output.

IV. PRELIMINARY STUDY

A. Implementing the Approach

We implemented a history slicing tool to realize the proposed technique. The technique targets Java projects managed by Git and handles dependencies on Java source code. We used JGit [18] to manipulate the change histories obtained from Git repositories. For extracting textual dependencies, we used CSlicer [2], which used the git-deps algorithm [19]. For extracting build dependencies, we used GumTree [20] to extract fine-grained changes from commits and find def-use relations in the obtained changes. We used Eclipse Java Development Tools [21] to search for the definition and use of identifiers.

B. Data Collection

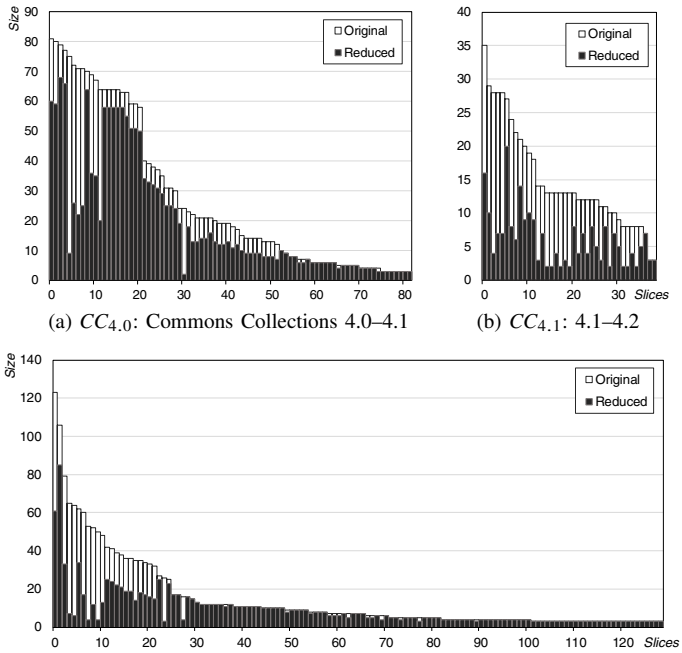
To answer the stated research question, we applied the implemented history slicing to several histories of open-source systems written in Java from the Apache Commons ecosystem. Table I shows the histories and systems we selected. The histories were selected as in-between two releases of the projects. The ends of this interval are termed first and last commits, as indicated in the table. For the simplicity of the input histories, we followed the selection criterion that there were no merge commits during the interval between two releases.

C. Data Analysis

We applied our proposed history slicing for each commit in the given history, i.e., each commit is regarded as the slicing criterion, and a history slice is obtained according to the slicing criterion. For example, because the history $CC_{4.0}$ consists of 211 commits, we obtained 211 slices, and each of them was a subset of the given original history.

TABLE II
ACCURACY IN DETECTING SYSTEMATIC EDITS

History	# Detected	# Correct	# Independent
CC_{all}	330 (82 + 248)	78 (78/82 = 95%)	73 (73/78 = 96%)
CN_{all}	146 (32 + 114)	31 (31/32 = 97%)	29 (29/31 = 94%)



(a) $CC_{4.0}$: Commons Collections 4.0–4.1

(b) $CC_{4.1}$: 4.1–4.2

(c) $CN_{3.3}$: Commons Net 3.3–3.4.

Fig. 3. Impact of systematic edits on history slices.

D. Results and Discussions

As a preliminary study, we investigated the accuracy of our systematic edits detection. The result is shown in Table II. We applied all the commits in Commons Collections (CC_{all}) and Commons Net (CN_{all}). Among the 2,971 and 1,954 commits in CC_{all} and CN_{all}, 330 and 146 were detected as systematic edits, respectively. Note that 82 of 330 and 32 of 146 commits included AST differences; the others only included white-space and comment changes. One of the authors manually confirmed whether these AST changes are actually systematic edits and if the edits in them are actually independent of each other. As a result, 78 and 31 of the detected systematic edits were correct, which suggests a detection precision of 95% and 97%, respectively. Furthermore, 73 and 29 of the correct systematic edits were regarded as splittable, i.e., each edit in them was independent of each other. This result suggests the validity of using systematic edits detected by our approach to split commits. Note that for two of the commits that are regarded as unsplittable, the necessary dependencies were covered by the build dependencies; thus, the resulting history slices were correct even though these non-splittable commits were regarded as systematic edits.

The impact of systematic edits on the size of history slices for each history are shown in Fig. 3. For each bar-plot, the bar shows a history slice, whose vertical size represents its size. All the slices are shown horizontally and are sorted in descending order by their size in the vertical. Slices whose sizes were less than three were excluded because they were too small to consider while discussing the reduction impact. The "Original" bars represent the slice size without applying dependency elimination with systematic edits detection, whereas the "Reduced" bars represent the size with the application of dependency elimination. Further, the numbers of detected sets of systematic edits are shown in the "# Systematic" column in Table I.

A reduction rate was observed in the size of the slices of $CC_{4.0}$, $CC_{4.1}$, and $CN_{3.3}$ by 20.7%, 57.2%, and 13.3% in average, respectively. The figure also shows that reduction succeeded at most slices but not for specific small subsets of them. This result suggests the importance of the consideration of systematic edits in the size reduction of history slices.

A history slice of $CN_{3.3}$ whose size was 123 obtained using the commit `ab2fd4f` as the slicing criterion was reduced to one whose size was 61 after enabling systematic edits detection. We confirmed the extent of change elements with widely spread dependence for this slice. We found that there were seven commits in the slice that textually depended on more than 10 commits. The severest commit was `4140189`, which had textual dependencies to 26 other commits. The elimination of dependencies did not allow the slicing criterion to reach these severe commits, which led to the size reduction of the resulting slice.

> The size of history slices decreased by 13.3–57.2% on average when splitting commits consisting of systematic edits.

V. CONCLUSION

This paper proposes an enhancement for history slicing based on the detection of systematic edits. By extracting the dependency relationships among change elements and eliminating some of them if they are detected as systematic edits, developers are able to obtain history slices of a reduced size. The conducted empirical studies show that the elimination of commit dependencies in the commits related to systematic edits reduces the size of resulting history slices by 13.3–57.2% on average.

As future work, several threats to the validity of the conducted empirical studies, e.g., generality of the selected projects and histories for the external validity, and the accuracy of the extracted dependencies for the internal validity, should be mitigated. Furthermore, we will further investigate the impact of commits consisting of independent sets of code by looking at the other types of changes.

ACKNOWLEDGMENTS

This work was partly supported by Kayamori Foundation of Informational Science Advancement and JSPS KAKENHI Grant Numbers JP18K11238, JP15K15970, JP15H02683, and JP15H02685.

REFERENCES

[1] Git SCM. [Online]. Available: https://git-scm.com/

[2] Y. Li, C. Zhu, J. Rubin, and M. Chechik, "Semantic slicing of software version histories," *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 182–201, 2018.

[3] B. Ray, M. Kim, S. Person, and N. Rungta, "Detecting and characterizing semantic inconsistencies in ported code," in *Proc. 28th IEEE/ACM International Conference on Automated Software Engineering*, 2013, pp. 367–377.

[4] F. Servant and J. A. Jones, "History slicing: Assisting code-evolution tasks," in *Proc. 20th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2012, pp. 43:1–43:11.

[5] ——, "Fuzzy fine-grained code-history analysis," in *Proc. 39th International Conference on Software Engineering*, 2017, pp. 746–757.

[6] K. Maruyama, T. Omori, and S. Hayashi, "Slicing fine-grained code change history," *IEICE Transactions on Information and Systems*, vol. E99-D, no. 3, pp. 671–687, 2016.

[7] M. Weiser, "Program slicing," *IEEE Transactions on Software Engineering*, vol. 10, no. 4, pp. 352–357, 1984.

[8] D. Kawrykow and M. P. Robillard, "Non-essential changes in version histories," in *Proc. 33rd International Conference on Software Engineering*, 2011, pp. 351–360.

[9] K. Herzig and A. Zeller, "The impact of tangled code changes," in *Proc. 10th Working Conference on Mining Software Repositories*, 2013, pp. 121–130.

[10] M. Barnett, C. Bird, J. a. Brunet, and S. K. Lahiri, "Helping developers help themselves: Automatic decomposition of code review changesets," in *Proc. 37th International Conference on Software Engineering*, 2015, pp. 134–144.

[11] H. Kirinuki, Y. Higo, K. Hotta, and S. Kusumoto, "Hey! Are you committing tangled changes?" in *Proc. 22nd International Conference on Program Comprehension*, 2014, pp. 262–265.

[12] S. Sothornprapakorn, S. Hayashi, and M. Saeki, "Visualizing a tangled change for supporting its decomposition and commit construction," in *Proc. 42nd IEEE Computer Software and Applications Conference*, 2018, pp. 74–79.

[13] C. Görg and P. Weißgerber, "Detecting and visualizing refactorings from software archives," in *Proc. 13th International Workshop on Program Comprehension*, 2005, pp. 205–214.

[14] J. Matsuda, S. Hayashi, and M. Saeki, "Hierarchical categorization of edit operations for separately committing large refactoring results," in *Proc. 14th International Workshop on Principles of Software Evolution*, 2015, pp. 19–27.

[15] M. Kim and D. Notkin, "Discovering and representing systematic code changes," in *Proc. 31st International Conference on Software Engineering*, 2009, pp. 309–319.

[16] N. Meng, M. Kim, and K. S. McKinley, "Systematic editing: Generating program transformations from an example," in *Proc. 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011, pp. 329–342.

[17] T. Molderez, R. Stevens, and C. D. Roover, "Mining change histories for unknown systematic edits," in *Proc. 14th International Conference on Mining Software Repositories*, 2017, pp. 248–256.

[18] JGit. [Online]. Available: https://www.eclipse.org/jgit/

[19] A. Spiers. git-deps. [Online]. Available: https://github.com/aspiers/git-deps

[20] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proc. 29th IEEE/ACM International Conference on Automated Software Engineering*, 2014, pp. 313–324.

[21] Eclipse Java development tools. [Online]. Available: https://www.eclipse.org/jdt/