Published version: Satoshi Yamashita, Shinpei Hayashi, Motoshi Saeki, "ChangeBeadsThreader: An Interactive Environment for Tailoring Automatically Untangled Changes,", in Proceedings of the 27th IEEE International Conference on Software Analysis, Evolution and Reengineering, pp. 657–661, 2020, https://doi.org/10.1109/SANER48275.2020.9054861

ChangeBeadsThreader: An Interactive Environment for Tailoring Automatically Untangled Changes

Satoshi Yamashita, Shinpei Hayashi, and Motoshi Saeki
School of Computing, Tokyo Institute of Technology, Tokyo 152–8552, Japan
Email: {yamashita, hayashi, saeki}@se.cs.titech.ac.jp

***Abstract*—To improve the usability of a revision history, change untangling, which reconstructs the history to ensure that changes contained in one commit are consistent, is important. Although there are several untangling approaches based on the clustering of fine-grained editing operations of source code, they often produce unsuitable result for a developer, and manual tailoring of the result is necessary. In this paper, we propose *ChangeBeadsThreader* (CBT), an interactive environment for splitting and merging change clusters to support the manual tailoring of untangled changes. CBT provides two features: 1) a two-dimensional space where fine-grained change history is visualized to help users find the clusters to be merged and 2) an augmented diff view that enables users to confirm the consistency of the changes in a specific cluster for finding those to be split. These features allow users to easily tailor automatically untangled changes.**

I. Introduction

Version control systems (VCSs) such as Git [1] are widely used in software development. Developers often mix changes in different intentional tasks in one commit, which results in a *tangled commit* [2]. Existing studies pointed out that tangled commits lead to difficulties in the code review [3], software evolution [4], and mining software repositories [5].

To prevent tangled commits, *change untangling*, i.e., decomposing tangled commits in a repository to ensure that each commit consists of changes of one intentional task, is important [2]. Existing change untangling approaches are classified into two types: 1) given a tangled commit, finding boundaries in the changes in the commit and separating them [5]–[9], and 2) given a sequence of finer-grained (edit-level) changes, clustering them and forming untangled commits based on the clusters [10], [11]. Although the latter approach requires a sequence of finer-grained changes rather than commits, it is efficient because it can capture the temporal characteristic of changes [10].

A challenge in automated change untangling is that providing *perfect* results is difficult. This is because the concept of *tasks*, which acts as a key role in commit construction, is ambiguous. Therefore, a manual tailoring process by developers is necessary to fix the result of an automated change untangling technique. Although Dias *et al.* provided a simple tool for fixing change untangling results [10], the tool did not focus on its usability, and the examinee pointed out issues due to the information overload when a large number of changes are given.

In this paper, we propose a tool named *ChangeBeadsThreader* (CBT), an interactive environment for splitting and merging change clusters to support the manual tailoring of untangled changes. CBT has two features: 1) a two-dimensional space where a fine-grained change history is visualized and 2) an augmented diff view that enables users to confirm the consistency of the changes in the specified clusters. The first feature helps us find the set of clusters to be merged whereas the second one helps us find changes to be split in a specific cluster. These features enable an efficient tailoring of untangled changes.

The remainder of this paper is organized as follows. Section II discusses related work. Section III explains the obstacles in tailoring untangled changes and defines the requirements for CBT. Section IV describes the detail of CBT. Section V concludes this paper and summarizes future work.

II. Related Work

Tangled commits lead to bad effects in software development. Bacchelli *et al.* [3] reported that tangled commits decrease the quality and increase the required time in code review. Herzig *et al.* [2] investigated the frequency and impact of tangled changes. They reported that more than 15% of bug-fix commits in five Java open source projects were tangled, which causes at least 16.5% of source files were incorrectly associated with bug reports.

To mitigate these issues, several approaches to untangle changes in a commit (the first type of untangling approaches) have been proposed [5]–[9]. For example, Barnett *et al.* [6] proposed a change untangling thechnique for C# programs. Their approach builds def-use dependency among methods and utilizes it for detecting related changes. A tool named ClusterChanges has been implemented to support the code review process for tangled commits.

Some approaches utilize fine-grained change history for change untangling. Dias *et al.* [10] proposed and implemented an automatic change untangling technique for fine-grained change history. They defined 13 metrics and investigated which metrics contribute to change untangling. As a result, the metrics of *timeDistance*, *numberOfEntriesDistance*, and *sameClass* were useful.

Several approaches to visualize fine-grained changes have been proposed. Yoon *et al.* [12], [13] proposed a tool named Azurite to support the selective undo operation with visualizing a sequence of fine-grained changes. Omori and Maruyama [14] proposed OperationReplayer to replay editing history for understanding software evolution. Barik *et al.* [15]

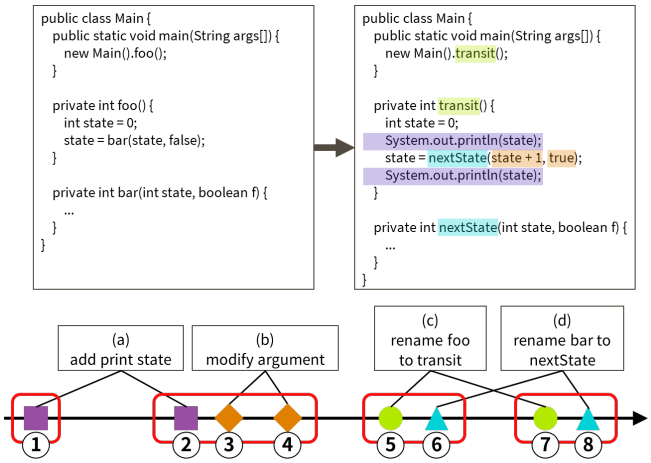

Fig. 1. Example of change untangling.

proposed a concept named Commit Bubbles, which enables developers to manipulate a commit history easily. However, to our knowledge, no tools have been proposed to support the tailoring process of commits automatically constructed from a fine-grained history.

III. Problems and Requirements

In this paper, we adopt the approach of merging a fine-grained change history obtained from change recording tools such as ChangeMacroRecorder [16] or Fluorite [17]. In this section, we explain the problems and requirements of change untangling for a fine-grained change history.

A. Running Example

Figure 1 shows an example of the change untangling. The source code before and after applying the changes are respectively shown in the upper left and the upper right of the figure. The changes (a) ① and ② add logging method invocations to show the value of `state`, (b) ③ and ④ fix the parameters of a method invocation, (c) ⑤ and ⑦ rename the method `foo` to `transit`, and (d) ⑥ and ⑧ rename the method `bar` to `nextState`. These changes are also shown in the timeline at the bottom of the figure.

By applying a clustering-based automated change untangling technique which utilize computed metric values on changes, we can have a separation from the given sequence of changes. The red rounded rectangles in the lower part of Fig. 1 show the clusters obtained from such a technique. In this result, the changes ②, ③, and ④ are incorrectly clustered because the executed time between the changes ② and ③ is close. The changes ⑤ and ⑥, and the changes ⑦ and ⑧ are also incorrectly identified as clusters in the same way.

It is difficult to construct a perfect automated change untangling technique because the concept of *tasks*, which acts as a key role in the commit construction, is ambiguous. Therefore, a manual tailoring process by developers is necessary to fix the result of an automated change untangling technique.

B. Challenges

To fix result, we need to tackle two problems. The first one is how to split clusters. In the tailoring process, we need to identify the changes to be excluded from a tangled cluster, e.g., the one consisting of ②, ③, and ④ in the previous example. One typical way to find such changes is to check the *diff* of all the changes in the cluster and find anomalies. In the case above, developers can identify that the changed line that adds a method invocation printing `state` (Line 10) should be excluded and be merged into another cluster. However, determining the changes to be excluded by associating the lines with the changes is a time-consuming and troublesome process, and developers may naively end up checking the diff of the three changes one by one.

The second problem is how to merge clusters. We need to merge two clusters if we find the changes in them belong to the same task; e.g., the new cluster consisting of ② obtained from the split above should be merged into the cluster consisting of ①. However, finding a cluster where the target cluster should be merged is also time-consuming because all the clusters except the target cluster can be the candidates, and developers need to check the changes in each cluster one by one. For example, developers need to discover the cluster for the changes to add invocations to print `state` (Line 10), i.e., the one consisting of ① by confirming the changes of each cluster. This process becomes hard if the number of changes becomes large.

To sum up, the following problems should be addressed in tailoring the automated results:

- checking the changes one by one in the target cluster to find the ones to be excluded and
- checking the clusters one by one to find the one that the target cluster should be merged.

C. Requirements

To solve these problems, this study needs to satisfy the following requirements. For splitting clusters, (a) it is required that the change which is the division candidate in the cluster can be visualized , and for the cluster merging operation, (b) it is required that the clusters with high merging possibility be visualized close to each other.

(a) allows developers to discover change to be split by checking diff of the cluster without checking individual changes. Also, (b) allows developers to search for a candidate to be merged based on proximity.

IV. ChangeBeadsThreader

A. Overview

We have developed a tool named ChangeBeads-Threader (CBT), which is an interactive environment for supporting the manual tailoring of automatically untangled changes[1]. In developing CBT, we conceptualized fine-grained

[1]A demonstration video is available at the following URL: https://www.dropbox.com/s/hlkn8idyp0x64xa/ChangeBeadsThreader-Demo.mp4

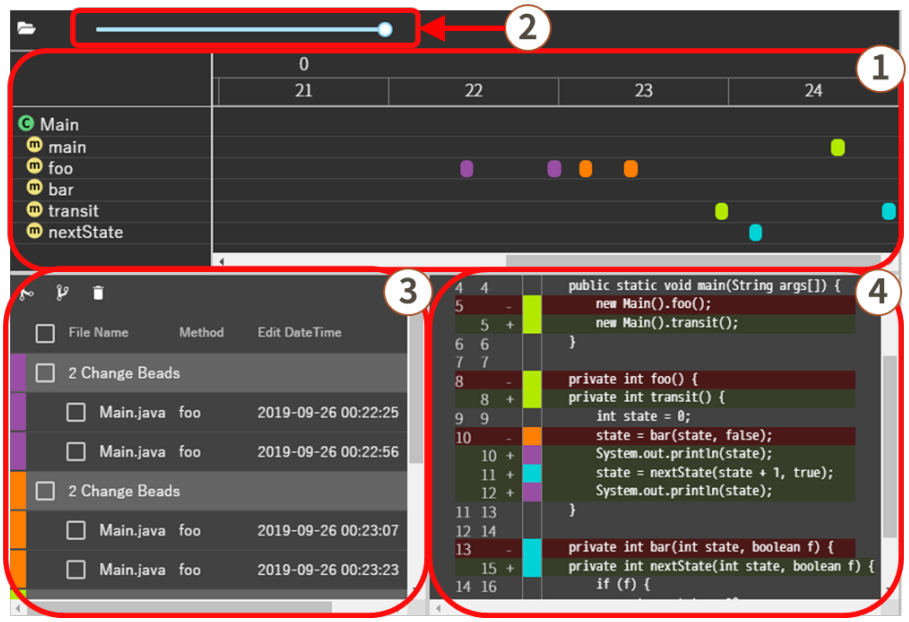

Fig. 2. Screenshot of ChangeBeadsThreader.

changes and and their clusters as beads and beadworks, respectively. The commit construction process is then regarded as beads threading. Hereafter, we call fine-grained changes treated in CBT *change beads*. In CBT, each change bead belongs to a specific cluster, and each cluster has its own color, which is automatically assigned by the tool. Change beads that belong to a cluster are rendered with the color of the cluster.

Figure 2 shows a screenshot of CBT, which consists of three panes. The *Map* pane (①) visualizes the change beads in the given history on the two-dimensional plane to assist the merging of change clusters. The scale of the timeline on the horizontal axis can be manipulated by operating the slider (②). The *List* and *Diff* panes (③ and ④) are for showing the information of the change beads of the selected clusters in Map pane. The List pane (③) lists the changes and shows their details such as the belonging cluster, the belonging class and method, and the time when it was committed. The Diff pane (④) shows an augmented *diff* of the changes, which allows users to check the consistency of the selected clusters.

Figure 3 shows the usage flow of CBT. CBT takes a Git repository consisting of fine-grained commits of a Java program as its input. It regards a sequence of commits as a sequence of fine-grained changes that are about to be committed as a single commit. Its output is another repository consisting of untangled commits, which are created by reordering and squashing the given fine-grained commits.

First, CBT preprocesses the input repository and applies an automated change untangling technique to build initial change clusters. The user sees the displayed clusters of change beads and finds one that is suspected as tangled. When selecting a cluster, its *diff* representation is displayed. The user splits it or merges it with another cluster until the user agrees with the result. Finally, the obtained clusters form untangled commits and stored into the resulting repository as an output.

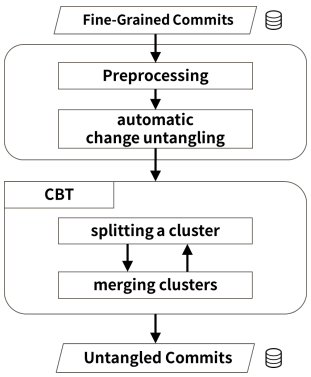

Fig. 3. Flowchart of ChangeBeadsThreader.

B. Preprocessing

As a preprocessing, CBT identifies some commits containing source files that their parsing fails to be excluded. They are squashed with their neighbor commits so that the changes made in them are still available in the repository. This is to avoid failing to identify the classes and methods that the changes in the commit belong to. Finally, CBT extracts the information of each commit.

C. Automatic Change Untangling

After the preprocessing, initial clusters are constructed by applying an automated change untangling technique. We followed the study by Dias *et al.* [10] on a clustering-based change untangling technique. They defined several metrics of fine-grained changes and applied a clustering algorithm using

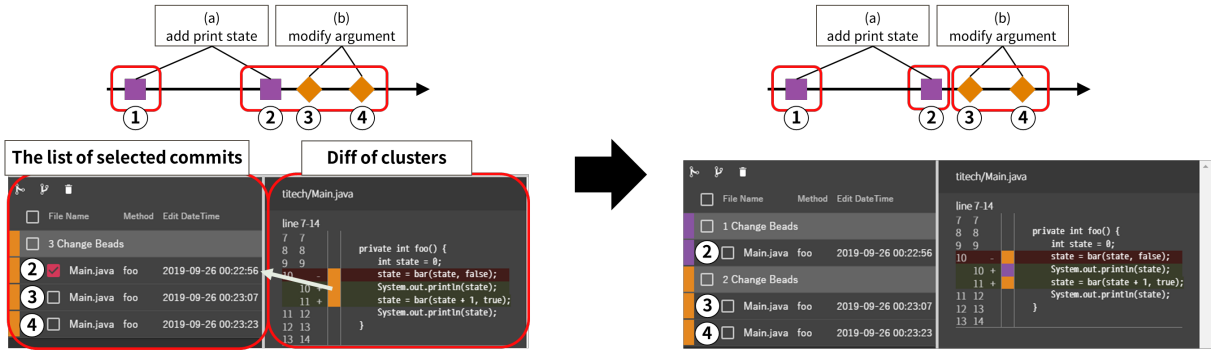

Fig. 4. Splitting a cluster.

TABLE I
METRICS FOR CALCULATING DISTANCE BETWEEN COMMITS

Name	Description
TimeDistance	Seconds between two given commits
NumberOfEntriesDistance	Number of commits in between two given commits
SameClass	Whether the changes in two given commits belong to the same class (0 or 1)
SameMethod	Whether the changes in two given commits belong to the same method (0 or 1)

the defined metrics. The initial clustering in CBT follows a simplified variant of their technique. We selected a subset of their metrics, as shown in Table I. The first two metrics quantify the temporal characteristic of changes whereas the other two metrics do the structural characteristic.

Using the metrics, CBT measures the distance for all the pairs of commits:

$$distance(c_1, c_2) = \sum_{i=1}^{|M|} \alpha_i M_i(c_1, c_2)$$

where c_1 and c_2 are the given commits, M is the set of metrics, $M_i(c_1, c_2)$ is the value of i-th metric for the given commits, and α_i is the weight of i-th metric. If the distance is lower than the threshold, they belong to the same cluster.

D. Splitting a Change Cluster

The Diff pane supports the split of clusters. Figure 4 shows an example cluster to be split. The left side of the figure shows the cluster before splitting, which consists of three changes: ②, ③, and ④, and here the user is trying to split it. All the changes in the cluster are listed in the List pane. Also, the changes are shown as the unified diff format in the Diff pane. Rather than a normal diff obtained from Git, the diff shown in the Diff pane is augmented with the clustering information; the colors of the belonging clusters are attached to the changed lines. On the left side of the figure, the orange color is attached to all three changed lines because all three changes belong to the same cluster.

The user found that the first and the last changed lines fix the parameters of the method `bar` to change the behavior whereas the middle changed line adds a new method invocation for a debugging purpose. From this observation, the user identified that the middle changed line has a different purpose from the other two changed lines and is an anomaly. Therefore, the user selects this anomalous change so that it is excluded from the cluster and forms a new cluster, as shown in the right side of the figure.

In this way, users can identify change clusters to be split without looking at the diff of each change separately.

E. Merging Change Clusters

The change beads visualization can be used to see all the changes at a glance and find candidate clusters where a cluster should be merged to. Figure 5 shows an example of the change beads visualization and how it can be utilized for the cluster merging. In this pane, the horizontal axis corresponds to the timeline representing *when* each change happened whereas the vertical axis corresponds to the locations representing *where* each change was performed. These two axes are based on the study by Dias *et al.* [10]; they showed that metrics based on temporal and structural distance are effective for clustering-based change untangling. We believe that these metrics are also effective in visualizing the distance between change beads so that they are adopted as the axes of our two-dimensional space. Change beads are displayed in this two-dimensional space in different colors according to their belonging cluster.

On the left side of Fig. 5, the user searches for the candidates, where the cluster consisting of the change ② should be merged, using the Map pane. Other than the cluster of ③ and ④, which is the origin of the newly introduced target cluster, the cluster of ① seems to have a change performed to the same method as ②, and the changes ① and ② are relatively closer in both the temporal and structural space; they were performed sequentially at the same method. As shown in the lower-left side of Fig. 5, the diff of all the changes in these two clusters is checked using the Diff pane as explained above. Since the user decides that the two clusters are identical, they are merged as shown in the right side of Fig. 5.

In this way, users can visually recognize clusters that are close in the temporal and the structural space, which helps users to find clusters where the target cluster should be merged to.

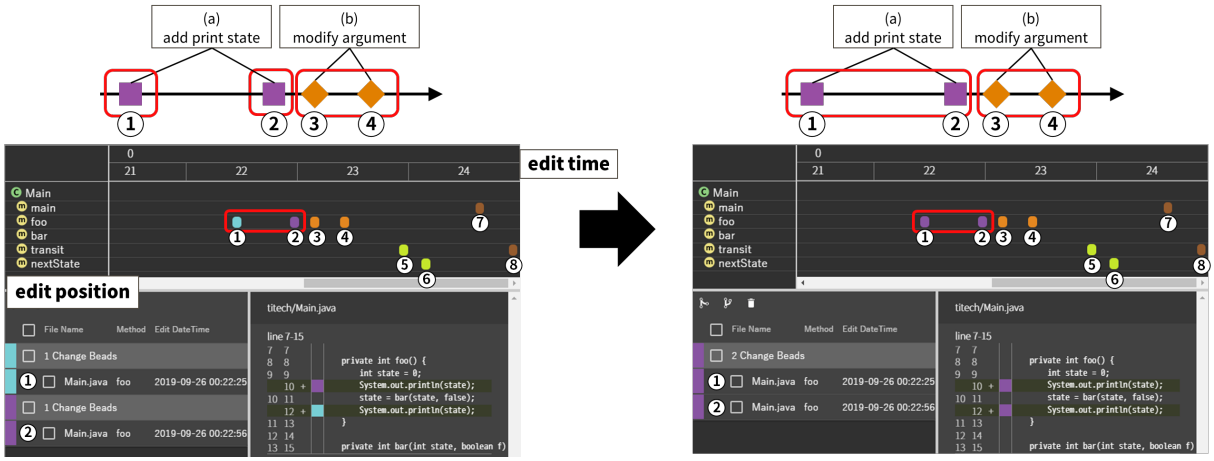

Fig. 5. Merging clusters.

F. Implementation

The implementation of CBT uses Kotlin as a development language and Eclipse Java development tools [18] and JGit [19] to retrieve change information. CBT itself is implemented as a desktop application using the Electron framework [20].

V. Conclusion

In this paper, we propose ChangeBeadsThreader, a tool to assist developers in tailoring automatic change untangling results. The tool provides the Diff pane that can confirm the consistency of changes within a specific cluster and the Map pane that visualizes fine-grained change history on a two-dimensional plane of the temporal and structural space. These panes support the discovery of the clusters to be merged and those to be split, which are the main task in tailoring untangled changes.

Our future work to improve the usability of CBT more is as follows:

- *New view* to visualize an overview of clusters to help users identify where they start to fix. To identify the cluster to be fixed first, outlining the abstracts of all clusters at a glance is necessary.
- *Highlighting change beads* that are close to those in the selected cluster for supporting more the cluster merging.
- *Visualizing metric values*. CBT uses metrics only for extracting temporal and structural characteristics. For example, visualizing def-use dependency of variables by drawing the connection between change beads should be beneficial because it helps users to find candidate change beads that should be merged but located at far.

Acknowledgments

This work was partly supported by JSPS Grants-in-Aid for Scientific Research Number JP18K11238.

References

[1] Git. [Online]. Available: https://git-scm.com/
[2] K. Herzig and A. Zeller, "The impact of tangled code changes," in *Proc. MSR*, 2013, pp. 121–130.
[3] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *Proc. ICSE*, 2013, pp. 712–721.
[4] S. Berczuk, *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*. Addison-Wesley, 2002.
[5] H. A. Nguyen, A. T. Nguyen, and T. N. Nguyen, "Filtering noise in mixed-purpose fixing commits to improve defect prediction and localization," in *Proc. ISSRE*, 2013, pp. 138–147.
[6] M. Barnett, C. Bird, J. Brunet, and S. K. Lahiri, "Helping developers help themselves: Automatic decomposition of code review changesets," in *Proc. ICSE*, vol. 1, 2015, pp. 134–144.
[7] W. Muylaert and C. De Roover, "Untangling composite commits using program slicing," in *Proc. SCAM*, 2018, pp. 193–202.
[8] P. Kreutzer, G. Dotzler, M. Ring, B. M. Eskofier, and M. Philippsen, "Automatic clustering of code changes," in *Proc. MSR*, 2016, pp. 61–72.
[9] S. Sothornprapakorn, S. Hayashi, and M. Saeki, "Visualizing a tangled change for supporting its decomposition and commit construction," in *Proc. COMPSAC*, 2018, pp. 74–79.
[10] M. Dias, A. Bacchelli, G. Gousios, D. Cassou, and S. Ducasse, "Untangling fine-grained code changes," in *Proc. SANER*, 2015, pp. 341–350.
[11] J. Matsuda, S. Hayashi, and M. Saeki, "Hierarchical categorization of edit operations for separately committing large refactoring results," in *Proc. IWPSE*, 2015, pp. 19–27.
[12] Y. Yoon, B. A. Myers, and S. Koo, "Visualization of fine-grained code change history," in *Proc. VL/HCC*, 2013, pp. 119–126.
[13] Y. Yoon and B. A. Myers, "Supporting selective undo in a code editor," in *Proc. ICSE*, vol. 1, 2015, pp. 223–233.
[14] T. Omori and K. Maruyama, "Identifying stagnation periods in software evolution by replaying editing operations," in *Proc. APSEC*, 2009, pp. 389–396.
[15] T. Barik, K. Lubick, and E. Murphy-Hill, "Commit bubbles," in *Proc. ICSE*, vol. 2, 2015, pp. 631–634.
[16] K. Maruyama, S. Hayashi, and T. Omori, "ChangeMacroRecorder: Recording fine-grained textual changes of source code," in *Proc. SANER*, March 2018, pp. 537–541.
[17] Y. Yoon and B. A. Myers, "Capturing and analyzing low-level events from the code editor," in *Proc. PLATEAU*, 2011, pp. 25–30.
[18] Eclipse Java development tools (JDT) — The Eclipse Foundation. [Online]. Available: https://www.eclipse.org/jdt/
[19] JGit — The Eclipse Foundation. [Online]. Available: https://www.eclipse.org/jgit/
[20] Electron — build cross platform desktop apps with javascript, html, and css. [Online]. Available: https://electronjs.org/