Published version: Sarocha Sothornprapakorn, Shinpei Hayashi, Motoshi Saeki, "Visualizing a Tangled Change for Supporting Its Decomposition and Commit Construction,", in Proceedings of the 42nd IEEE Computer Software and Applications Conference, pp. 74–79, 2018, https://doi.org/10.1109/COMPSAC.2018.00018

Visualizing a Tangled Change for Supporting Its Decomposition and Commit Construction

Sarocha Sothornprapakorn, Shinpei Hayashi, and Motoshi Saeki
Tokyo Institute of Technology, Tokyo 152–8550, Japan
{sarocha,hayashi,saeki}@se.cs.titech.ac.jp

***Abstract*—Developers often save multiple kinds of source code edits into a commit in a version control system, producing a tangled change, which is difficult to understand and revert. However, its separation using an existing sequence-based change representation is tough. We propose a new visualization technique to show the details of a tangled change and align its component edits in a tree structure for expressing multiple groups of changes. Our technique is combined with utilizing refactoring detection and change relevance calculation techniques for constructing the structural tree. Our combination allows us to divide the change into several associations. We have implemented a tool and conducted a controlled experiment with industrial developers to confirm its usefulness and efficiency. Results show that by using our tool with tree visualization, the subjects could understand and decompose tangled changes easier, faster, and higher accuracy than the baseline file list visualization.**

***Index Terms*—tangled change, refactoring, visualization**

I. Introduction

Developers usually save their edits in a version control system as a *task level commit* [1], i.e., a commit containing edits related with only one task, which is regarded as a good practice because it makes changes easy to understand [2] and revert [3]. There are many task intentions such as implementing a new feature, fixing a bug, and improving the quality of source code structure by applying a refactoring [4].

However, some investigations show that developers sometimes commit edits related with multiple task completions at once, and it results in a *tangled change* [5] This is regarded as a bad habit since it often causes more complicated commits, which make developers hard to recognize their related tasks.

Many researchers tried to find a reliable way to decompose a tangled change into small task level commits [6]–[8]. However, tangled change decomposition is still tough to break it down using traditional *diff* format, which displays file list-based edit chunks consisting of the added and removed lines. We think that this visualization is not suitable for large tangled changes including many edits through many files.

We propose an approach to decompose a tangled change to construct proper task level commits. Our technique visualizes a tangled change in a tree structure including its grouping information of multiple types which enables us to apply multiple grouping criteria. We utilize refactoring detection and relevance calculation techniques for organizing edits as multiple groups. We have also implemented an interactive tool to display the detail of a tangled change, which enables developers to reorganize task level commits from a tangled change with the guidance of the tree visualization. By using our technique, developers can efficiently understand the structure of the tangled change, quickly untangle it, and simply produce multiple commits from it.

There are three main contributions in this paper.

1) A new technique to visualize a tangled change as a tree structure which enables users to handle multiple grouping criteria. To the best of our knowledge, this is the first work to provide a tree visualization for displaying the information from various kinds of grouping criteria.
2) A semi-automated interactive tool with utilizing refactoring detection and relevance calculation techniques for the decomposition of a tangled change.
3) A controlled experiment consisting of eight industrial developers to prove the efficiency of our visualization concept and evaluate the tool.

The rest of the paper is organized as follows. Section II introduces the proposed technique and its sub steps. The tool implementation and its evaluation with a controlled experiment are described in Sections III and IV, respectively. The summary of related work is in Section V. Finally, Section VI concludes this paper.

II. Proposed Technique

Figure 1 represents the overall process flow of our technique. Firstly, we obtain before (the old) and after (the new) versions of a tangled change as inputs. Then, we create *diff* patches of the tangled change. Secondly, we utilize a refactoring detection technique to separate the refactoring part with the non-refactoring part. Also, we utilize relevance calculation technique to know which lines are related, especially in the part of non-refactoring changes (Section II-A). As a result, we can group the associated lines together and construct multiple edits based on the relevance as a tree hierarchy. Finally, we connect refactoring-based with relevance-based results to generate our informative tree representing the detail of the given tangled change (Section II-B). Note that the connection process takes into consideration that refactoring edits may also be grouped with non-refactoring edits in some cases.

We visualize those arrangements using informative tree-based structure to represent the tangled change composition. This tree helps developers to recognize the relationship between each line. They can simply choose some parts of the tree to be a new task level commit.

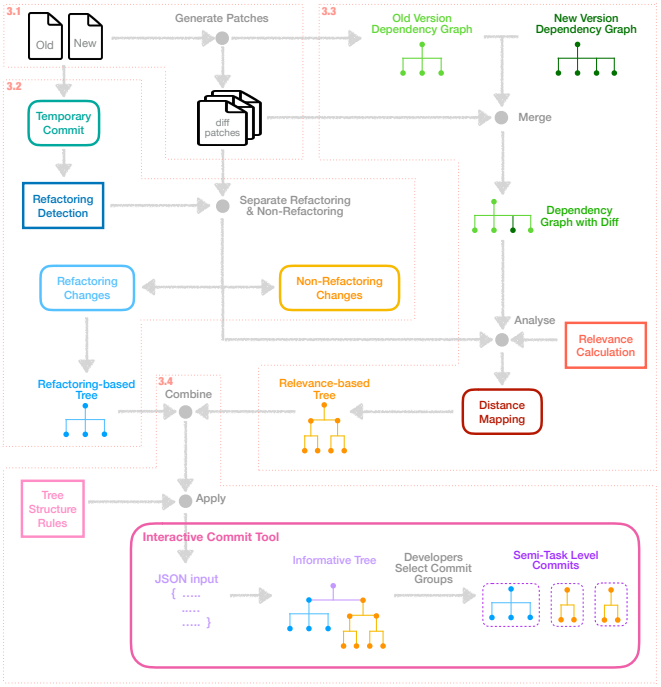

Fig. 1. Overview of the proposed technique.

Hello.java

```
public class SimpleCode {
    public static void main(String args[]) {
-       printA();
-       printB();
-       printC();
+       printAA();
+       printBB();
+       printCC();
+       Hello h = new Hello("Tua");
+       h.printHello();
    }
-   public static void printA() {
+   public static void printAA() {
        System.out.println("Hello World A!");
    }
-   public static void printB() {
-       System.out.println("Hello World B!");
+   public static void printBB() {
+       System.out.println("Hello World BB!");
    }
-   public static void printC() {
+   public static void printCC() {
        System.out.println("Hello World C!");
    }
}
```

SimpleCode.java

```
public class Hello {
    private String name;
-
+   public Hello(String name) {
+       this.name = name;
+   }
+   public void printHello() {
+       System.out.println("Hello " + name);
+   }
}
```

Fig. 2. Running example in *git diff* patch representation.

A running example of a tangled change will be used throughout the process that we explain in the next subsections. Its *git diff* patch representation is shown in Figure 2. It involves the following change tasks of three different types:

1) Feature enhancement: Print "Hello" following by name,
2) Bug fixing: Change "Hello World B!" to "Hello World BB!", and
3) Refactoring: Rename Method from *printA* to *printAA*, *printB* to *printBB*, and *printC* to *printCC*.

We utilize *git diff* command to generate the diff patches. A diff patch consists of all the change information happened in a file. Thus, the number of the generated patches is the same as the number of the modified files.

A. Refactoring Detection and Relevance Calculation

Separating the refactoring changes, which are the behavior preserving part, out of the non-refactoring changes, which are behavior modification part, is needed in our technique. Furthermore, some refactoring operations consist of other refactoring operations. To detect the performed refactoring operations, we utilized an existing refactoring detection tool called *RefactoringMiner* [9]. We chose *RefactoringMiner* because it is not only easy to use but also accurate to detect refactorings [10]. In the case of the running example shown in Figure 2, three *Rename Method* refactoring operations are detected: Rename of the method from *printA* to *printAA*, *printB* to *printBB*, and *printC* to *printCC*.

Another information that we used is the dependency between each line of changed code for untangling a change to group relevant lines together. We capture the degree how much are program elements associated regarding their dependency relation. We utilized transitive data dependency information. We used the distance in the interprocedural program dependence graph to capture the strength of each dependency. The distance ranges from 1 to infinity; the less number means a stronger relation between elements.

B. Tree Formation and Tool Visualization

We combined the results obtained from the previous step to construct a tree. We use the tree as an input for our interactive tool, and developers can select some parts of the tree to be committed in the Git directory. The main idea of the tree structure is to display the overall information at the higher level of the tree and to give the related specific changes in detail as the lower tree nodes of the tree.

As for the refactoring results, we assign a tree node for each detected refactoring operation. Each refactoring operation instance is categorized by its type, and a parent-child relationship between the type node and the instance node is defined. Then, the nodes of the change lines caused by a refactoring operation belong to the refactoring operation node. In addition, for a compositional refactoring, the composition relationship between refactoring operations is translated as a parent-child relationship in the tree. Finally, all the refactoring subtrees are collected and categorized by the refactoring root node. According to the running example in Section II, a refactoring tree is constructed as shown in Figure 3.

Similarly, a relevance-based subtree is generated using the non-refactoring edit chunks using the calculated relevance. We group lines together if the distance between them is less than the pre-defined threshold. However, this threshold is also used for grouping refactoring part and non-refactoring part such as related rename according to a feature enhancement which we think these two changes should be committed together even it breaks a task level definition. Such exception can be regarded as a heuristic condition of our grouping rule. Considering this situation, our tree can represent multiple criteria by duplicating tree nodes. Some sub-refactoring nodes can also be copied into another related sub-refactoring depending on its distance.

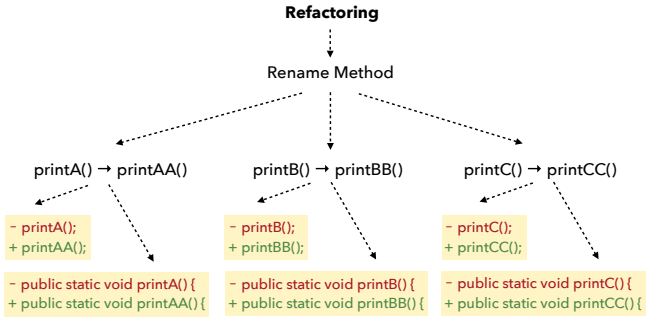

Fig. 3. Refactoring subtree of motivation example.

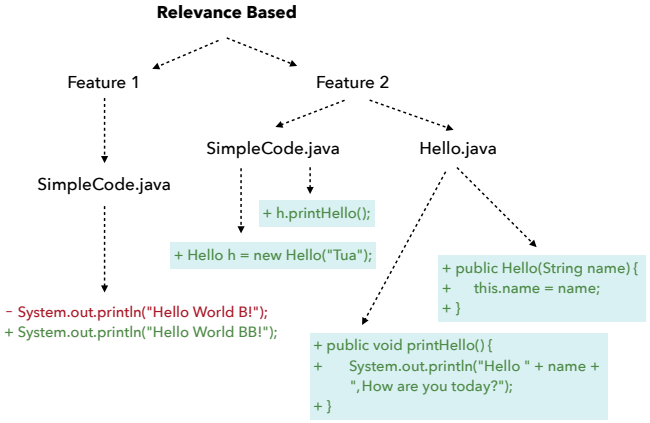

Fig. 4. Relevance-based subtree of motivation example.

We can construct a relevance-based subtree for the running example in Section II as shown in Figure 4.

The final informative tree result is generated by combining refactoring and non-refactoring subtrees. The left panel of the tool screenshot shown in Figure 5 shows the final tree for the motivating example.

III. Tool Implementation

We have implemented a proof-of-concept tool named *ChTree* as a web application. A screenshot of the tool is shown in Figure 5. Users can select some lines of a change and create

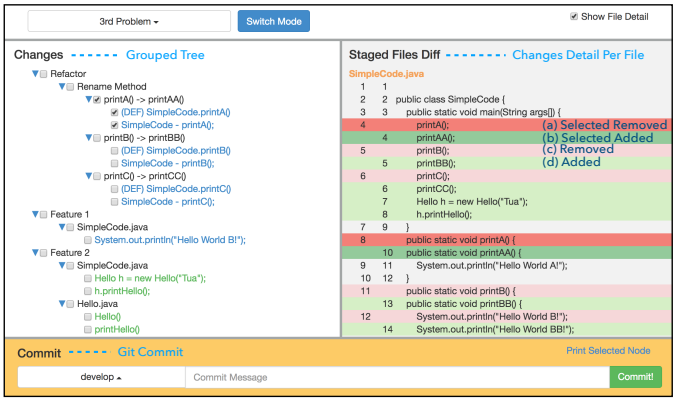

Fig. 5. Example of tool screenshot.

TABLE I
Examples of Code Change in Address Book

	Task Type	Short description
E_1	Feature Enhancement	Let user find a contact not only using name keyword but also using address keyword
	Bug Fixing	Fix the description of *ViewAll* command
	Refactoring	Extract Superclass *Contact* from *Address, Email, Phone* classes
	Refactoring	Rename Method from *getPersonsWithNameContainingAnyKeyword* to *getPersonsContainingAnyKeyword*
E_2	Feature Enhancement	Let user find a contact by using insensitive keyword from name instead of case sensitive keyword
	Bug Fixing	Fix the example of *Email* class
	Refactoring	Extract Interface *Printable* from *Name, Phone, Email, Address* classes
	Refactoring	Rename Class from *TypicalPersons* to *TypicalPeople*

a Git commit from the selected lines by using the *Commit* panel at the bottom. The left panel of the tool presents an organized informative tree. Users can select a small part of the changed file from the left panel, and then related lines of code with the selected item are displayed in the right panel. The right panel shows change information including the selection state for each line of code in each file. Removed lines are highlighted in red (a, c) whereas added lines are highlighted in green (b, d). More saturated red and green colors (a, b) are for removed and added but selected lines, respectively.

We empirically defined the distance value of 4 as the threshold of the relevance.

IV. Evaluation

We conducted a controlled experiment to confirm that *ChTree*, our technique, supports the decomposition and understanding of a tangled change by answering the following research questions (RQs). These RQs can verify that *ChTree* is more suitable for tangled change decomposition than the baseline using ordinary file list-based visualization.

- **RQ 1 (Understandability):** *Does ChTree make developers easier to understand changes than ChList?*
- **RQ 2 (Ease of Commit Construction):** *Does ChTree make developers easier to separate changes than ChList?*
- **RQ 3 (Usability):** *Does ChTree's usability good enough to be used in the developers' Git workflow?*

ChList, the baseline tool, was also implemented for the comparison purpose. It is a customized version of *ChTree* whose tree part simply shows the list of the modified files. Since users of *ChList* obtain less information than *ChTree*, we gave to them additional information: the list of detected refactorings and the strength of the relevance between changed lines.

A. Experimental Setup

We selected the address book application [11], a public and educational Java application, to be our code base for this experiment. We think that it is complex enough; it contains a sufficiently large number of lines of code including a proper level of difficulty to understand and recognize the functionality in a short time during the experiment.

We prepared two examples of code change at the same level for inspecting both the proposed technique and the baseline. Their level can be calibrated by its difficulty, lines of code changes, the number of the changed file, and the number of each type of the tasks implemented for each type. Table I shows the examples detail. We also prepared another easy example to be used in the practice session before the experiment.

As for creating the *ground-truth* groups of commits for each example, we followed a basic strategy: non-behavioral (refactoring and comments) changes and behavioral (non-refactoring) changes should be separated. However, since such changes are often related in some cases, we allowed some useful exceptions. Hence, we finally used the following rules:

1) refactoring and non-refactoring changes should be committed *separately*,
2) renamed changes caused by a new feature addition can be committed *together* with the feature addition,
3) code comments (JavaDoc) can be committed *together* with related changes,
4) non-refactoring changes of different tasks or irrelevant tasks should be committed *separately*,
5) unit test changes can be committed *together* with the related non-refactoring changes, and
6) refactoring changes which happen for applying another refactoring should be committed *together*.

In the direction of examining our technique against the baseline, we have to make sure about an equivalent level of information that subjects obtain. Hence, we have to provide additional information to the participant who was running the treatment on the baseline approach includes the list of the detected refactoring operations came from RefactoringMiner presented in a PDF file. Also, a list of distance mapping between each line change was given in a spreadsheet.

Finally, we obtained eight participants to cover all the possible combinations of the two examples and two visualization modes (*ChTree* and *ChList*), whose programming and Git experiences range 5–18 and 1–5 years, respectively. The experiment was done with a randomly ordered combination.

We deployed tangled change inputs into our proof-of-concept tool. After instructing how our tool works and the basic knowledge of the address book code base using the practice example, participants worked for two treatments. Each treatment consists of an example (either of E_1 or E_2) and a tool mode (*ChTree* or *ChList*). In a treatment, a participant first read the given change using the given tool and understand its content (Understanding). The participant then asked the number of tasks related to the change and the purpose of each task. If his/her answer differed from the ground truth, the operator described the truth. Then, the participant was asked to decompose the given change and construct commits using the given tool (Commit Construction). The time of the understanding and the commit construction were measured. Finally, the participant was asked to fill out his/her opinion to the prepared questionnaire form.

TABLE II
EXPERIMENTAL RESULTS

		Time (min)			Accuracy		
	Treatment	Und.	Com.	Total	*P*	*R*	*RI*
P_1	E_1 *ChTree*	10	14	24	1.00	0.98	0.99
	E_2 *ChList*	8	16	24	0.87	0.58	0.81
P_2	E_1 *ChList*	9	10	19	1.00	1.00	1.00
	E_2 *ChTree*	10	7	17	1.00	0.83	0.94
P_3	E_2 *ChTree*	14	9	23	1.00	0.83	0.94
	E_1 *ChList*	9	6	15	0.78	0.93	0.84
P_4	E_2 *ChList*	36	13	49	1.00	0.81	0.93
	E_1 *ChTree*	15	5	20	1.00	1.00	1.00
P_5	E_1 *ChTree*	10	9	19	1.00	1.00	1.00
	E_2 *ChList*	4	8	12	0.89	0.99	0.95
P_6	E_1 *ChList*	14	10	24	1.00	0.97	0.98
	E_2 *ChTree*	12	5	17	1.00	0.83	0.94
P_7	E_2 *ChTree*	11	8	19	0.94	0.99	0.97
	E_1 *ChList*	8	10	18	0.88	0.94	0.91
P_8	E_2 *ChList*	9	7	16	1.00	1.00	1.00
	E_1 *ChTree*	8	2	10	1.00	1.00	1.00
	Ave. *ChTree*	11.25	7.38	18.63	0.99	0.94	0.97
	Ave. *ChList*	12.13	10.00	22.13	0.92	0.91	0.93

B. RQ 1: Understandability

1) Experimental Data Analysis: Weighing the understandability of *ChTree* and *ChList* has several ways. We used three approaches: averaging time spent on understanding each problem in each mode, questionnaire opinion based on a 5-level Likert scale in *Grouping* section, and open-end comments including the conversation during the experiment execution.

2) Results and Discussion: First, we calculated the average time for each mode and compared the time. Our consumption is if *ChTree* makes developers easier to understand tangled changes than *ChList*, *ChTree* should use less amount of time than *ChList*.

The time spent for change understanding is shown in Table II. The average time used for recognizing tangled changes in *ChTree* was 11.25 min, which was less than the average time applied for recognizing in *ChList*, which was 12.13 min.

Second, we analyzed the obtained opinion scales and inspected various interesting points. There were six of eight participants agreed or strongly agreed on the understandability of *ChTree* while there were only three participants agreed or strongly agreed on *ChList*.

Third, we recorded the screen and conversation occurred during the experimental accomplishment. The participant P_1 extremely mentioned about the tree usefulness. However, the participant P_4 was not sure about the grouping accuracy, so he did not use the tree structure when understanding the changes. This ignorance makes the time of P_4 spending on *ChTree*, which is 15 min, greater than P_1, which used only 10 min to recognize the overall objectives of E_1.

Also, some participants mentioned their thought that the grouping feature in *ChTree* is more suitable for not only the code commit separation but also the code review session or code debugging. This opinion indicated that the participants agreed to the usefulness in term of change understandability.

C. RQ 2: Ease of Commit Construction

1) Experimental Data Analysis: We used four strategies for measuring the ability of commit construction from a tangled change with *ChTree* against *ChList*. The approaches are the average time spent for committing each change in each mode according to the commit policy, the correctness of the constructed commits comparing with the pre-defined ground truths, questions as opinion based on a 5-level Likert scale in *Grouping* section, and open-end comments along with the conversation during the experiment execution.

In accordance with the calculation of the correctness of the constructed commits, we utilize the evaluation way of clustering algorithms. We list all pairs of leaf node from the tree on both two examples. E_1 contains 41 leaf nodes, so the number of node pairs is 820. Meanwhile, since E_2 contains 40 leaf nodes, the number of node pairs is 780. We categorized the obtained pairs as true positive (*TP*), true negative (*TN*), false positive (*FP*), and false negative (*FN*) based on the relevant pairs in the ground truth. Then, we computed the precision (*P*), recall (*R*) and rand index (*RI*) for each mode by using below formulas to verify the clustering quality:

$$P = \frac{TP}{TP + FP},\ R = \frac{TP}{TP + FN},\ RI = \frac{TP + TN}{TP + FP + FN + TN}.$$

We combined the results of two examples and averaged the results because we believe that both are in the same difficulty level and that it was appropriate to unify the results without considering the order of execution and the example item.

2) Results and Discussion: We measured the average time spent on tangled changes committing as shown in Table II. Participants used 7.38 min to produce the qualified commits in *ChTree* but 10 min in *ChList*. The data represented that there was a high potential that *ChTree* could help to reduce the time used in committing changes.

After interpreting the gathered opinion scale questions, the results designated that seven of eight participants chose the feature *Groups are showing in tree form* as a feature that they like in *ChTree*. Furthermore, 75% of candidates also agreed or strongly agreed on the grouping feature usefulness. Meanwhile, in *ChList*, six of eight participants selected *Select some line on each file* as a feature that they desire. Only half of the participants liked the feature *List of changed files are showing in list form.* Conversely, 87.5% of candidates disagreed or strongly disagreed on the grouping feature usefulness of *ChList*. This observation somehow evidences the advantages of *ChTree* over *ChList* on tangled change decomposition.

According to the collected comments, a participant mentioned about *ChTree* that it was easy to know the related files and refactoring in every single change. Another participant also said that *ChTree* "*helped me to group all the related task easily.*"

Table II also represents accuracy results for each participant. The precision of grouping by *ChTree* was 0.99 whereas that of *ChList* was 0.92. One of the arguments which decrease the precision value is an action that a participant grouped irrelevant tasks together in one commit. This error happened because of overlooking small changed part in a file where major changes were done in another purpose even though the participant identified the small change in code changes during the understanding session beforehand. However, representing multiple changes in one file using *ChList* can produce some missing of attention. This kind of error frequently occurred when candidates used *ChList* and happened only once for *ChTree*.

We asked a participant who grouped irrelevant changes together about his grouping reason. He said that he did not recognize the change detail carefully and also did not read the tree label cautiously. Thus, he decided everything by himself without persuaded by *ChTree*. This situation indicates that users require a time to be familiar with *ChTree* usability. Thus, we concluded that *ChTree* was better than *ChList* with regards to the time spent and the accuracy of the commit construction.

According to the small reduction of recall value in both two modes (0.94 and 0.91), we found that most of the participants constructed finer-grained commits than the ground truth. Despite the commit policy specified that there are some exceptions, e.g., a rename refactoring and related non-refactoring changes should be committed together, participants somehow ignored this policy. In addition, in the part of non-refactoring changes, there were some changes applied to variables that were out of the scope of the distance mapping with the actually related changes. Hence, we constructed it separately in *ChTree*. A distinction of this kind of relevant changes reduced the recall value because the participants somehow influenced by the provided tree structure.

D. RQ 3: Usability

1) Experimental Data Analysis: In order to answer about the usability of the tool, we investigated the results of the inquiries in opinion based on a 5-level Likert scale in the section *Usability*. Also, we gathered the observations during experiments and open-end comments in the filled form.

2) Results and Discussion: The collected opinion scale information denoted that 50% of participants agreed or strongly agreed to mainly use *ChTree* in their current Git workflow. On the other hand, no one agreed or strongly agreed to mainly use *ChList* in their current Git workflow.

Nonetheless, while experimenting, we got many useful comments from the participants about numerous way to improve the usability of the tools. However, there are various positive comments from open-end questions in the experiment questionnaire. They signified that the usability of *ChTree* was good to use in Git workflow. One participant mentioned that "*it is really useful for developers. I hope that it will be released soon. This is what I am looking for a lifetime.*" Meanwhile, other two participants stated that the tool was "*easy to use*" and "*good usability*" while another said that "*the idea is good and it's helpful.*"

E. Threats to Validity

Internal validity: We run the experiment with eight participants to cover all four possible combinations of two visualization modes and two examples. The order of the treatments was

randomized to prevent the priority bias. We also measured the overall time spent to see the tendency in case that a participant utilized the understanding time for deciding how the commit should be generated. However, the format of the given tangled change examples might contain be biased.

External validity: In the correctness measurement on RQ 2, the results depended on not only the understanding level but also the granularity sense of each developer. Eight results are too less to confirm our comparing in term of statistical significance. Thus, we cannot strengthen that our approach is completely better than the baseline. However, we can conclude that our informative tree visualization tends to outperform the general file list visualization from the support of the open-end comments and the scale of opinions that we obtained from all participants.

V. Related Work

Zhang et al. [12] proposed *CRITICS*, an Eclipse plug-in for investigating systematic changes during code review session by presenting similar changes and detecting potential mistakes from overlooking points. Conversely, our approach focuses on supporting developers during their pre-committing phase and assisting them to decompose multi-purposes changes.

Matsuda et al. [13] introduced a technique for change rearranging by grouping fine-grained changes based on commit policy criteria by focusing on refactoring-related changes to configure the composition of commits. Our tool focuses on both refactoring and non-refactoring relation and also allows developers to organize and generate commits by using the interaction of our tool.

Steinert et al. [14] proposed a tool named *CoExist* that contains automated decomposition, grouping, and commit construction features of Smalltalk programming language by recording fine-grained changes and automate making commits as snapshots including the meta information in a background process. In our approach, we differently support Java programming language and use coarse-grained changes from *git diff* command beside letting users decide what and when to create a commit.

Tauemel et al. [15] also presented an interactive tool called *Thresher* to support developers for decomposing and grouping tangled changes using tree-based information and let developers manually adjust the organizations and construct commits. The main idea of this approach is similar to our approach, but there are several different procedures. They considered the relation of changes by analyzing activities from recorded fine-grained operations and also utilized *CoExist* [14] technique for deciding the relevance of activities. Conversely, our technique not only investigates performed refactoring operations but also utilizes the program dependence to calculate the change relevance. Their refactoring detection can detect only the refactoring operations performed using the IDE but in our approach manually applied refactoring operations are detectable.

VI. Conclusion

A tangled change consisting of multiple task purposes is often saved in a version control system as a single commit. Using a file-based list resulted from *git diff* command is not suitable for tangled change representation. Our approach decomposes a tangled change by utilizing refactoring detection and relevance calculation techniques. It displays the groups of changes as an informative tree to describe the tangled change structure which well informs our multiple grouping criteria. We have developed an interactive tool that enables developers to customize their commits so they can express their intention how the commits should be organized. We conducted a controlled experiment to confirm the usefulness and measured the efficiency of task level commit generation comparing between tree-based visualization (our approach) and file list-based visualization (baseline). By using our tool, our subjects could understand, classify, and decompose the tangled changes easier and faster than the baseline. The familiarity of the traditional approach reduced the time gap between the changes understanding. However, the time spent in separating a tangled change into task level commits was obviously reduced from 10 to 7.38 min on average.

Acknowledgments

This work was partly supported by JSPS KAKENHI Numbers JP15K15970, JP15H02683, and JP15H02685.

References

[1] S. P. Berczuk and B. Appleton, *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*. Addison-Wesley, 2002.

[2] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim, "How do software engineers understand code changes?: An exploratory study in industry," in *Proc. FSE*, 2012, pp. 51:1–51:11.

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

[4] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[5] K. Herzig and A. Zeller, "The impact of tangled code changes," in *Proc. MSR*, 2013, pp. 121–130.

[6] H. Kirinuki, Y. Higo, K. Hotta, and S. Kusumoto, "Hey! are you committing tangled changes?" in *Proc. ICPC*, 2014, pp. 262–265.

[7] M. Barnett, C. Bird, J. a. Brunet, and S. K. Lahiri, "Helping developers help themselves: Automatic decomposition of code review changesets," in *Proc. ICSE*, 2015, pp. 134–144.

[8] M. Dias, A. Bacchelli, G. Gousios, D. Cassou, and S. Ducasse, "Untangling fine-grained code changes," in *Proc. SANER*, 2015, pp. 341–350.

[9] D. Silva, N. Tsantalis, and M. T. Valente, "Why we refactor? confessions of github contributors," in *Proc. FSE*, 2016, pp. 858–870.

[10] N. Tsantalis, M. Mansouri, L. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *Proc. ICSE*, 2018.

[11] D. C. R. Leow Yijin, "Command line interface address book application," https://github.com/se-edu/addressbook-level2.

[12] T. Zhang, M. Song, J. Pinedo, and M. Kim, "Interactive code review for systematic changes," in *Proc. ICSE*, 2015, pp. 111–122.

[13] 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.

[14] B. Steinert, D. Cassou, and R. Hirschfeld, "CoExist: Overcoming aversion to change," in *Proc. DLS*, 2012, pp. 107–118.

[15] M. Taeumel, S. Platz, B. Steinert, R. Hirschfeld, and H. Masuhara, "Unravel programming sessions with THRESHER: Identifying coherent and complete sets of fine-granular source code changes," *Computer Software*, vol. 34, no. 1, pp. 103–118, 2017.