Hey! Are You Committing Tangled Changes?

Hiroyuki Kirinuki, Yoshiki Higo, Keisuke Hotta, Shinji Kusumoto
Osaka University
1-5 Yamadaoka, Suita, Osaka, Japan
{h-kirink,higo,k-hotta,kusumoto}@ist.osaka-u.ac.jp

ABSTRACT

Although there is a principle that states a commit should only include changes for a single task, it is not always respected by developers. This means that code repositories often include commits that contain tangled changes. The presence of such tangled changes hinders analyzing code repositories because most mining software repository (MSR) approaches are designed with the assumption that every commit includes only changes for a single task. In this paper, we propose a technique to inform developers that they are in the process of committing tangled changes. The proposed technique utilizes the changes included in the past commits to judge whether a given commit includes tangled changes. If it determines that the proposed commit may include tangled changes, it offers suggestions on how the tangled changes can be split into a set of untangled changes.

Categories and Subject Descriptors

D.2.2 [**Design Tools and Techniques**]: Computer-aided software engineering; D.2.7 [**Distribution, Maintenance, and Enhancement**]: Version control

General Terms

Management

Keywords

Understanding commits, Mining software repositories, Tangled changes

1. INTRODUCTION

Recently, research areas related to mining software repositories (MSR) have been very active and are attracting significant attention [5, 13]. A software repository includes a variety of historical information on the past activities related to the software itself, while an MSR contains actions, techniques, and methodologies for extracting and deriving useful knowledge that can be used in future development.

ICPC'14, June 2–3, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2879-1/14/06...$15.00
http://dx.doi.org/10.1145/2597008.2597798

Historical code repositories such as CVS, SVN, and Git are often-mined repositories because they are rich sources of information that can help developers gain knowledge or principles that will be useful for software engineering tasks [6, 17]. Therefore, numerous researchers have been studying methods for mining historical code repositories. Such research efforts are based in a wide variety of interests, including defects [2, 6, 8, 15, 17, 22], changes [1, 3, 4, 14, 21, 23], and clones [10, 20].

However, despite the generally accepted principle that a commit should include only changes for a single task, programmers perform commits that include *tangled changes*, which are a set of changes for completing two or more tasks [9]. Besides, Murphy-Hill et al. revealed that changes for refactorings are often committed with changes for other tasks [18]. Nguyen et al. reported 11 – 39% of bug fixing commits include other changes [19].

Commits that include tangled changes can become obstacles to the efficient activities of other software developments [7]. Some of those obstacles are presented below:

- First, tangled changes are inappropriate for merging the code of one branch to another. Merging code means that the modifications on a branch in the past are applied to the code of another branch. If a commit includes only those changes that we want to merge, it should be possible to perform code-merging operations efficiently. However, if a commit includes tangled changes, it is difficult to merge only the changes related to the specified task. In such cases, we need to choose one of the two following workarounds:
  - manual merging operations without using the merging function of the version control systems, or
  - manual cancelling operations for the merged extra code after using the merging function.

  However, when either choice is selected, it is necessary to perform manual operations.
- Second, change reversions for a specified task cannot easily be performed on tangled change commits. When the reverting function of a version control system is used, all the changes in the commit are reverted.
- Third, it is difficult to review delta code on a commit that includes tangled changes. This means we have to seek the changes for the specified task in the commit.

In addition to the various software development difficulties mentioned above, the presence of tangled changes also

hinders analyzing code repositories because most MSR approaches are designed for code repositories that only include untangled changes. If code repositories include tangled changes, the accuracy of MSR analyses drops. A good example of such cases can be found in analyses used to identify evolutionary couplings [21]. An evolutionary coupling usually refers to a set of software modules, such as source files, that were changed together. In many approaches, two or more source files are regarded as having an evolutionary coupling if they were changed in the same commit. Accordingly, methods of identifying such evolutionary couplings are obviously designed for code repositories that only include untangled changes.

Many MSR approaches have preprocessing operations that are used to eliminate tangled changes. In such cases, large commits often become targets for elimination. Herein, a large commit refers to a commit that includes numerous changed lines or a large number of changed source files. However, while eliminating large commits can decrease the negative impacts of tangled changes, such preprocessing is less than perfect [16].

- Untangled changes can be found in large commits. Therefore, eliminating all large commits will result in the unintentional elimination of such untangled changes.
- Tangled changes are sometimes included in small commits, and will not be eliminated by preprocessing that only target large commits.

In order to avoid the problems caused by tangled changes, we are researching techniques for avoiding the presence of tangled changes in code repositories. Currently, our approach aims at warning developers that they are in the process of committing tangled changes. This technique is very beneficial to them because they can be made aware of potential tangled changes before committing them.

While our research is still in the early stage, in this paper, we show (1) the key idea behind our design approach for avoiding tangled changes, and (2) experimental results performed on open source software conducted with a naive implementation of the approach. The contributions of this paper are as follows:

- It addresses the importance of untangled changes in software development activities and code repositories analyses. It also advances the argument that avoiding committing tangled changes is a reasonable way to achieve tangled-change-free repositories.
- It proposes a technique for identifying whether the changes that a developer is about to commit are tangled. If the given changes are identified as tangled, the technique also suggests how they can be split into a set of untangled changes.
- It reports a small experiment conducted on an open source project to confirm its utility. It also provides a discussion aimed at increasing the detection accuracy of tangled changes.

2. PROPOSED TECHNIQUE

The key idea behind the proposed technique involves utilizing code change patterns. Our research group has revealed

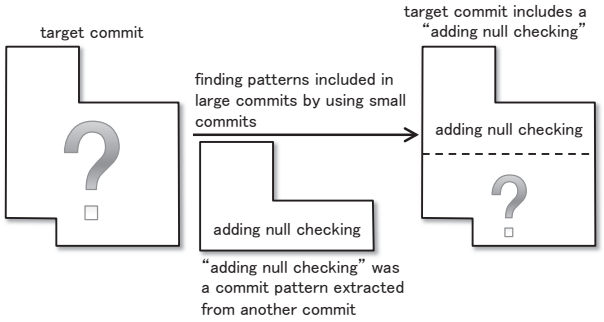

Figure 1: Key idea behind the proposed technique

that some code change patterns appear repeatedly in different commits [11]. Figure 1 shows an example of this key idea. If we have a commit "*adding null checking*", we can utilize it to split different commits.

The proposed technique is intended for use just before a developer commits her/his changes to the repository. It consists of two procedures, *creating a database of templates* and *checking whether the developer's changes are tangled*.

In this explanation, we assume that there are already n revisions in the code repository, and a developer is ready to commit her/his changes.

2.1 Creating a Database of Templates

Changes are distilled from every pair of two consecutive revisions. Figure 2 shows the distillation procedure, which consists of the following three steps:

STEP 1: Source files changed between revisions r and $r+1$ $(1 \leq r \leq n-1)$ are parsed and a list of program statements is extracted. Each program statement consists of a token sequence.

STEP 2: The two program statement lists are compared with the longest common subsequent algorithm and differences between them are identified. Each of the differences is called a **pattern**, which consists of pre-code and post-code. If the difference is a code addition, its pre-code is empty. Similarly, if the difference is a code deletion, its post-code is empty.

STEP 3: Variable names in the patterns are replaced with special tokens. Literals and other user-defined variables such as method names are not replaced.

A pattern set identified from each pair of two consecutive revision is a **template**. All the identified templates are stored in a database.

2.2 Checking Whether the Developer's changes are tangled

The proposed technique determines whether a developer's changes by using the template database. First, the developer's changes are extracted by comparing source files in the developer's working directory with the head revision in the code repository. Then, the proposed technique checks to see whether any of the templates in the database are subsets of her/his changes.

- If there is no template that is a subset of the changes, the proposed technique does not regard the changes as

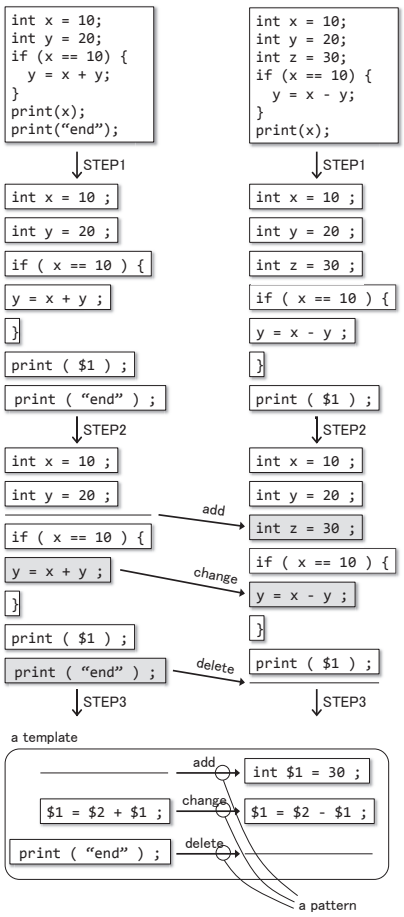

Figure 2: Steps of the proposed technique

tangled. The developer can then commit the changes with confidence that they are only for a single task.

- If there is a template that is a subset of the changes, the proposed technique provides feedback to the developer. Using that feedback, she or he then determines whether the changes are actually tangled. If she/he determines that the changes are tangled, she/he will be able to commit the tangled changes separately.

3. CASE STUDY

In order to investigate how often the proposed technique identifies tangled changes, we conducted an experiment on open source software, jEdit. We used the first 2,000 revisions as our target because manual verification was required to determine whether each of the identified positives were actually tangled changes. The speed of the proposed technique is very quick, it took less than a second to judge whether each commit includes tangled changes or not.

Specifically, we checked whether changes between the two revisions r and $r+1$ were tangled by using the templates extracted from the previous revisions. Each of the commits that the proposed technique regarded as potentially tangled were then checked manually. The proposed technique identified 63 commits as possible candidates for containing tangled changes. Table 1 summarizes the results.

Logging and *condition checking* such as null checking were often included in the suspected tangled changes. Figure 3 shows an example of the suspected tangled changes that were detected. In this commit, there are two kinds of changes: (1) shifting to single-thread parsing from a multi-thread one, and (2) adding if-statement for checking whether the buffer is loaded. The log of this commit is "*fix bad bug*". The commit log refers only to the buffer checking. This indicates that, when a developer committed the changes, he/she forgot that two kinds of changes were being performed. Checking whether the changes being readied for commitment are tangled is a reasonable level of support for such a developer. Additionally, one-stop checking before committing changes allows code repositories to avoid including tangled changes.

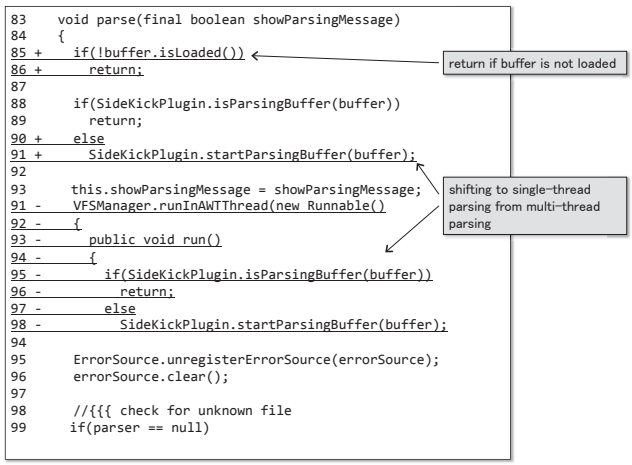

Figure 3: Detected tangled changes

In table 1, we can see that various kinds of changes can become tangled. The refactorings, such as *rename* and *code move*, that were included in tangled changes, are called *floss* refactorings [18]. There are two commits that included *adding "else return null"*. This change is used for handling unexpected situations. The aforementioned condition checking and this error handling are the kinds of bugs that occur when developers forget to write code to handle unexpected situations. These often-occurring error handlings are also sometimes included in tangled changes.

On the other hand, a number of the candidate tangled

Table 1: Manual investigation result (each number in a cell refers to the number of commits categorized in the specified modification type)

Modification Type	T	M	U	Total
Logging	7	2	0	9
Condition checking	12	5	4	21
GUI	1	0	0	1
Field declaration	0	4	0	4
Method declaration	1	0	0	1
Local variable initialization	0	1	13	14
Code move (refactoring)	2	0	0	2
Rename (refactoring)	2	0	0	2
Boolean value substitution	0	1	6	7
Adding "else return null"	2	0	0	2
Total	27	13	23	63

T: Tangled, M: Maybe, U: Untangled

changes turned out to be false positives. Many of these were categorized into *local variable initialization* and *boolean value substitution*. Since variable declaration statements and substitution statements are often-used instructions in the Java programming language, our pattern-based approach tends to identify them as tangled changes, even if they are not. The authors think there are two ways to prevent such false positives:

- The first way is to utilize data dependencies between the instructions and other changed instructions. If they are connected with other changed instructions, they are not to be regarded as tangle changes.
- The second way is to utilize repeated code structures. In the Java programming language, there are many repeated instructions such as switch-case statements, else-if statements, and repeated variable declaration statements [12]. If two or more templates are included in a repeated code, they are related to each other, even if they have no data dependence.

With the above heuristics, it should be possible to eliminate most of the false positives detected in our experiments.

We also have to mention the correctness of created templates. Currently, the proposed technique leverages all the changes in the past commits for creating templates. However, in such a way, tangled changes in the past commits are treated as untangled ones. We need to seek a methodology to make templates from only untangled changes.

4. CONCLUSION

In this paper, we discussed how tangled changes can hinder software development and code repository analysis. Then, as a method for preventing tangled changes, we proposed a technique to identify whether a developer's changes are tangled. Using the technique, developers can be made aware that their changes are potentially tangled and can be given the opportunity to commit the tangled changes separately.

We applied the proposed technique to open source software, jEdit. Currently, our technique is limited to naive implementation that does not include any heuristics. The experimental result showed that almost half of the identified changes were actually tangled, but that the others were not. We also discussed how we could improve the proposed technique based on the experimental result and suggested two heuristics that can be used to eliminate false positives.

In the future, we intend to implement our system with other techniques that will improve its tangled change detection accuracy. Two potential techniques involve using data dependencies between program statements and considering repeated structures in source code.

5. ACKNOWLEDGMENT

This work was supported by MEXT/JSPS KAKENHI 25220003, 24650011, and 24680002.

6. REFERENCES

[1] M. Askari and R. Holt. Information theoretic evaluation of change prediction models for large-scale software. In *Proc. of MSR2006*.

[2] C. Boogerd and L. Moonen. Evaluating the Relation between Coding Standard Violations and Faults within and across Software Versions. In *Proc. of MSR2009*.

[3] G. Canfora, L. Cerulo, and M. D. Penta. Identifying Changed Source Code Lines from Version Repositories. In *Proc. of MSR2007*.

[4] E. Giger, M. Pinzger, and H. C. Gall. Can We Predict Types of Code Changes? An Empirical Analysis. In *Proc. of MSR2012*.

[5] A. E. Hassan. The Road Ahead for Mining Software Repositories. In *Proc. of ICSM2008*.

[6] H. Hata, O. Mizuno, and T. Kikuno. Bug Prediction Based on Fine-Grained Module Histories. In *Proc. of ICSE2012*.

[7] S. Hayashi and M. Saeki. Recording finer-grained software evolution with ide: an annotation-based approach. In *Proc. of IWPSE-EVOL2010*.

[8] K. Herzig, S. Just, and A. Zeller.

[9] K. Herzig and A. Zeller. The impact of tangled code changes. In *Proc. of MSR2013*.

[10] Y. Higo, K. Hotta, and S. Kusumoto. Enhancement of crd-based clone tracking. In *Proc. of IWPSE2013*.

[11] Y. Higo and S. Kusumoto. How often do unintended incosistencies happend? –deriving modification patterns and detecting overlooked code fragments–. In *Proc. of ICSM2012*.

[12] A. Imazato, yui Sasaki, Y. Higo, and S. Kusumoto. Improving process of source code modification focusing on repeated code. In *Proc. of PROFES2013*.

[13] H. Kadgi, M. L. Collard, and J. I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Wiley JSME2007*.

[14] H. Kagdi, J. I. Maletic, and B. Sharif. Mining software repositories for traceability links. In *Proc. of ICPC2007*.

[15] Y. Kamei, S. Matsumoto, A. Monden, K. Matsumoto, B. Adams, and A. E. Hassan. Revisiting Common Bug Prediction Findings Using Effort-Aware Models. In *Proc. of ICSM2010*.

[16] N. Kusunoki, K. Hotta, Y. Higo, and S. Kusumoto. How much do code repositories include peripheral modifications? In *Proc. of IWESEP2013*.

[17] R. Moser, W. Pedrycz, and G. Succi. A Comparative Analysis of the Efficiency of Change Metrics and Static Code Attributes for Defect Prediction. In *Proc. of ICSE2008*.

[18] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. *IEEE TSE2012*.

[19] 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. of ISSRE2013*.

[20] F. Rahman, C. Bird, and P. Devanbu. Clones: What is that Smell? In *Proc. of MSR2010*.

[21] A. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting Source Code Changes by Mining Change History. *IEEE TSE2004*.

[22] H. Zhang. An Investigation of the Relationships between Lines of Code and Defects. In *Proc. of ICSM2009*.

[23] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining Version Histories to Guide Software Changes. *IEEE TSE2005*.