Investigation and Detection of Split Commit

Ryo Arima, Yoshiki Higo, and Shinji Kusumoto
Graduate School of Information Science and Technology, Osaka University, Japan
{r-arima, higo, kusumoto}@ist.osaka-u.ac.jp

***Abstract*—Each commit in repositories of version control systems should include code changes for only a single task. However, in real repositories, there are many commits for multiple tasks and tasks split into multiple commits. We call the latter split commits. In this research, we firstly investigate how many and what kinds of split commits are included in repositories. Then, we classify the found split commits into three categories. Based on the classification, we propose a new technique to detect split commits automatically. This is the first research that proposes a technique to detect split commits. To evaluate the proposed technique, we apply it to repositories of two open source software. The results show that the proposed technique detects split commits with high accuracy (precision is 0.8 and F-measure is 0.7).**

***Index Terms*—Version control system, Split commit.**

I. Introduction

When we use version control systems for software development, a meaningful set of code changes such as a bug-fix or a functional addition should be a single commit. In this research, we call such a meaningful set of code changes *task*. Previous research studies revealed that commits including multiple tasks, which they call *tangled commits*, have negative impacts on performance of repository analyses [1]. There are also several research studies proposing techniques to split tangled commits into different appropriate commits [1] [2]. However, there is no research on merging commits into which a task is split as a single appropriate commit. In this research, we call such a pair of commits containing a split task *split commit*. The followings are main issues caused by the presence of split commits.

- Performance decrement of repository analyses such as an evolutional coupling [3][4][5].
- Understandability decrement of past commits.

The contributions of this research are as follows:

- investigating how many and what kinds of split commits are included in repositories, and
- proposing a new technique to find split commits automatically.

II. Investigation of Split commit

As the first step of this research, we investigated how many and what kinds of split commits existed in development histories. In this research, we defined that a pair of commits c_1 and c_2 is a *split commit* if both the commits contain the same task. If there are more then two commits containing the same task, every pair in the commits is a split commit.

A. Investigation approach

The targets of this investigation are 100 commits in two repositories, Apache Commons Collection[1] and Retrofit[2]. We manually checked whether every pair in these commits was a split commit or not.

B. Results

We found that 81 split commits were included in both repositories by checking 1,714 commit pairs. By investigating characteristics of the 81 split commits in more detail, we categorized them with the following three-level classification.

1) Level-1 (snippet level changes): In Level-1, c_2 need not be a single independent commit at all. For example, adding some changes for overlooked code fragments or reverting previous changes are Level-1. In Level-1, c_1 and c_2 change the same or very closely located code.

2) Level-2 (method level changes): In Level-2, the changes in c_2 depend on the changes in c_1. In most cases of Level-2, two methods having a calling or an inheritance relationship are changed in c_1 and c_2. For example, a commit adding a new method and a commit adding calling the method to another method are Level-2 split commit.

3) Level-3 (function level changes): In Level-3, changes in c_1 and c_2 are adding, deleting or modifying functions of the same class or module. For example, a commit adding a getter method for a member of a class and a commit adding a setter method for the same member form a Level-3 split commit.

Table I shows the number of the detected split commits from the viewpoint of this classification. Based on the above features of each level split commit, we propose an automatic detection technique of split commits in the next section.

III. Our technique for Split commit detection

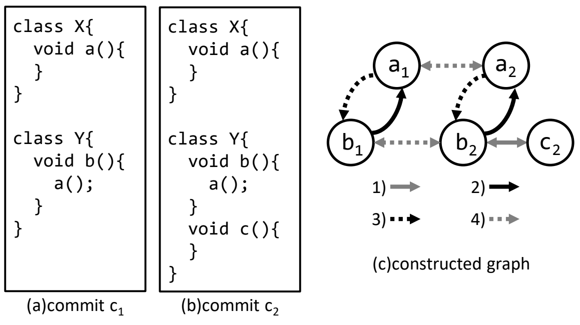
Our technique takes a pair of commits as input, and it outputs whether the pair is a split commit or not.

First, our technique constructs a graph from source code at each commit of the input automatically. Each vertex in the graph represents a method in the source code. Method m in

TABLE I
Detected split commits

	Level-1		Level-2		Level-3	
Apache	13	(0.9%)	21	(1.4%)	49	(3.3%)
Retrofit	4	(1.7%)	9	(3.9%)	32	(14.0%)

[1]https://github.com/apache/commons-collections
[2]https://github.com/square/retrofit



```
class X{
  void a(){
  }
}

class Y{
  void b(){
    a();
  }
}
```

```
class X{
  void a(){
  }
}

class Y{
  void b(){
    a();
  }
  void c(){
  }
}
```

Fig. 1. The graph constructed from source code.

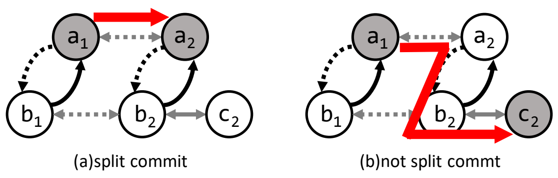


Fig. 2. Distinguish split commits.

commit c_n is represented as m_n in this paper. Each edge in the graph means that their methods have either of the following relationships.

1) The two methods in the two commits are the same.
2) The method calls the other method.
3) The method is called by the other method.
4) The methods are defined in the same class.

The proposed technique constructs a graph shown in Figure 1(c) from two commits shown in Figures 1(a) and 1(b).

Then, the proposed technique determines whether the given two commits are a split commit or not based on the distance between vertices representing modified methods. If the distance between modified methods is shorter than the threshold, the technique regards the input as a split commit. In Figure 2, highlighted vertices represent modified methods, and thick arrows represent the shortest paths from a modified method to another nearest modified method. A graph shown in Figure 2(a) is a split commit because the distance between modified methods is short, whereas a graph shown in Figure 2(b) is not a split commit because the distance between modified methods is long.

TABLE II
RESULTS OF THE EXPERIMENT.

	Target 1			Target 2
	precision	recall	F-measure	precision
Collections	0.714	0.714	0.714	0.822
Retrofit	1.000	0.594	0.745	0.884

IV. EXPERIMENTATION

To evaluate our proposed technique, we conducted experiments on repositories of open source software.

A. Targets

We applied our proposed technique to two targets. Target 1 is 1,714 pairs of commits obtained by the investigation of the two repositories in Section II. As evaluation measures of this experiment, precision, recall and F-measure were calculated by comparing the output of the proposed technique with the results of the investigation.

Target 2 is 18,619 pairs of commits in the same repository as the investigation described in Section II. By checking manually whether each pair regarded as a split commit by the technique was correct or not, precision was calculated.

B. Results

Table II shows the results. F-measure of target 1 was 0.7, and precision of target 2 was 0.7. Our proposed technique is useful for split commits detection.

In cases where the technique detected split commits by mistake, there were pairs of commits that methods modified at many commits were modified at both commits. For example, entry points of software such as main methods are modified at many commits.

V. CONCLUSION

In this research, we manually checked two repositories of open source software and found 81 split commits. We also classified them into three categories. Then, we proposed an automatic detection technique of split commits with the heuristics that we had derived from the manual checking. In our evaluation, F-measure and precision of the proposed technique are 0.7 and 0.8, respectively.

In the future, we are going to investigate the effect of split commits on performance of repository analyses.

ACKNOWLEDGEMNT

This work was supported by JSPS KAKENHI Grant Number JP25220003.

REFERENCES

[1] K. Herzig and A. Zeller, "The Impact of Tangled Code Changes," in *Proc. of the 10th International Workshop on Mining Software Repositories*, 2013, pp. 121–130.
[2] H. Kirinuki, Y. Higo, K. Hotta, and S. Kusumoto, "Hey! are you committing tangled changes?" in *Proc. of the 22nd International Conference on Program Comprehension*, 2014, pp. 262–265.
[3] J. M. Bieman, A. A. Andrews, and H. J. Yang, "Understanding change-proneness in OO software through visualization," in *Proc. of the 11th International Workshop on Program Comprehension*, 2003, pp. 44–53.
[4] H. Gall, K. Hajek, and M. Jazayeri, "Detection of logical coupling based on product release history," in *Proc. of the 1998 International Conference on Software Maintenance*, 1998, pp. 190–198.
[5] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," *IEEE Trans. Softw. Eng.*, vol. 31(6), pp. 429–445, 2005.