How We Resolve Conflict: An Empirical Study of Method-Level Conflict Resolution

Ryohei Yuzuki*, Hideaki Hata*, Kenichi Matsumoto*
*Graduate School of Information Science
Nara Institute of Science and Technology, Japan
Email: {yuzuki.ryohei.yg6, hata, matumoto}@is.naist.jp

***Abstract*—Context: Branching and merging are common activities in large-scale software development projects. Isolated development with branching enables developers to focus their effort on their specific tasks without wasting time on the problems caused by other developers' changes. After the completion of tasks in branches, such branches should be integrated into common branches by merging. When conflicts occur in merging, developers need to resolve the conflicts, which are troublesome. Goal: To support conflict resolution in merging, we aim to understand how conflicts are resolved in practice from large-scale study. Method: We present techniques for identifying conflicts and detecting conflict resolution in method level. Result: From the analysis of 10 OSS projects written in Java, we found that (1) 44% (339/779) of conflicts are caused by changing concurrently the same positions of methods, 48% (375/779) are by deleting methods, 8% (65/779) are by renaming methods, and that (2) 99% (771/779) of conflicts are resolved by adopting one method directly. Conclusions: Our results suggest that most of conflicts are resolved by simple way. One of our future works is developing methods for supporting conflict resolution.**

I. Introduction

Branching enables individual developers or developer teams to work on their tasks separately from other development. Changes in branches will be integrated into other branches by merging the multiple latest commits. Even if bug fixing or feature implementation has been completed on a branch, the task is not considered complete until the change has reached the other, especially main (master, trunk, or root), branch successfully.

Although branching has become easy using the recent distributed version control systems like Git, there remain difficulties in merging. Merges have multiple parents, which makes it difficult to understand changes and identify origins. Because of these difficulties, resolving conflicts and verifying changes are troublesome. In addition, large merging, which include many conflicts should be especially high risk.

Phillips et al. conducted a survey composed of questions about branching and merging for project administrators, then they obtained the result where 54% of the respondents think that the most significant problem about merging is conflict [11]. It is required that we need to know the activity of conflict resolution to discuss or suggest the means of supporting or automation of conflict resolution. However, the way how to resolve conflicts at method-level in practice is unrevealed. Our goal is to survey how conflicts are resolved in real projects.

The structure of this paper is as follows. First, we introduce related work in Section II, and our approach in Section III. Second, we explain about target OSS projects to be surveyed in Section IV, and obtained results in Section V. Finally, we discuss in Section VI, and conclude in Section VII.

II. Related Work

There are various studies about conflict detection and reduction [1]–[3], [9]. For example, Apel suggested Semistructured Merge, which is independent on programming language and reduces syntax and semantic errors [1].

Resolution is also one of the most significant problem on conflict [9] However, previous studies before distributed version control system becomes the mainstream are mainly on model-level. There are few empirical studies about conflict resolution at method-level.

Phillips conducted a survey composed of 21 questions for about 300 project administrators to reveal how developers treat branching and merging and what is the successful merging strategy in practice [11]. They found purpose of use and problem about branching and merging, and suggested current status of merging on projects and hypothesisis of successful parallel devdelopment strategy.

Nan defined Cost-benefit model using dependence pair which is generated by Semantic Diff and effect of code change by global variables and ranked method resolution priorities when conflicts are detected on multi methods [6], [10].

These studies don't mention the method-level knowledge and concrete resolution action. If the way how developers changed conflicting method is revealed, we can discuss about optimal methods of conflict resolution.

In addition, there is a very interesting case on automatic bug fixing. Weimer fixed program bugs automatically using genetic programming [8], [12]. However, their method can generate nonsensical patches by random code mutations. Therefore, Kim inspected more than 60000 human-written patches and obtained common fix patterns. Then he fixed program bugs using pattern-based approach [7]. Kim obtained more reasonable bug fixing than Weimer's by leveraging human-knowledge. In the same way, if human-knowledge is applyed to conflict resolution, it is possible to obtain high-quality automation.

III. Approach

We analyze with the following procedure to see how developers resolved conflicts.

1) Finding a merge commit.
2) Finding a pair of conflict commits.
3) Finding conflicted methods in conflict commits.
4) Detecting the causes of conflicts.
5) Detecting the resolutions of conflicts.

A. Merge Commits

A merging commit has at least 2 parent commits. Therefore, it is enough to obtain such commits as merging commits.

B. Conflict Commits

We think that conflict commits are the commits in branches that have completed each task of branches. It is natural to consider the parents of the merge commits as the pair of conflict commits since they are actually merged. However it is often the case that developers create additional commits before merging to resolve conflicts step by step. In such case, the parent commits are not the original conflict commits.

Among many pairs of commits between two branches, we assume that conflict commits are the pair with the largest differences since the differences may increase during the task completion phases, and the differences may decrease during the merge preparation phases. In this study, we measure the number of different methods as the differences.

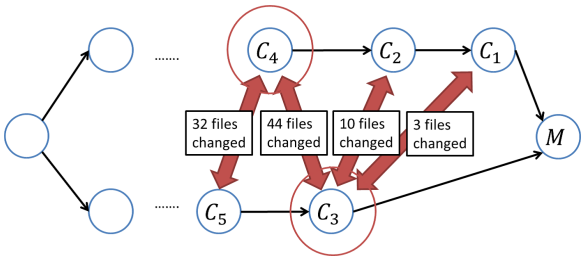

Fig. 1. The differences in the pair of commits

We obtain two commits whose difference of files is largest as candidates, and try to merge them actually. For example, in Fig. 1, C_3 and C_4 are candidates because their difference of files is the largest (44 files changed).

C. Conflicted Methods

If we operate merging with the identified conflict commits, Git will list up conflict files. In this study we adopt Historage [5], fine-grained version control system, to analyze histories. Since Historage repositories store methods as files, method-level histories can be analyzed with similar to file-level histories.

D. Causes

Git can lists up the different files between two commits. Differences have 3 attributes—ADD, RENAME, and DELETE, which means added file, renamed file, deleted file. We compare the differences between merging commits (M), their conflict commits (C_1, C_2), and their original commits (O). The causes can be summarized as follows:

- **CHANGE SAME POSITION** Same position of the same methods are modified.
- **DELETION** One of methods are deleted in one conflict commit.
- **RENAMING** One of methods are renamed in one conflict commit.

E. Resolution

Based on analyzing the contents of conflicted methods, the resolution can be summarized as follows:

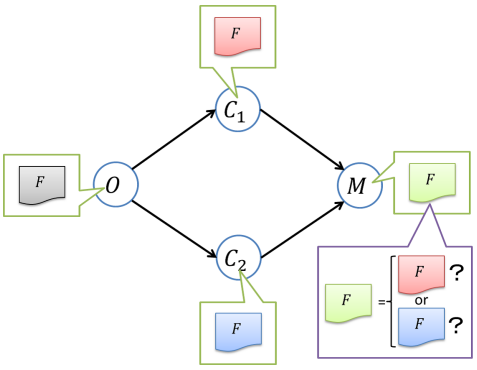

Fig. 2. 1-WAY (medhod exists in 3 commits)

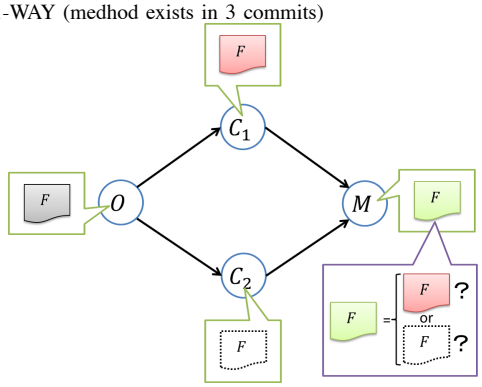

Fig. 3. 1-WAY (medhod exists in 1 or 2 commits)

- **1-WAY** The conflict is resolved by adopting one of contents in conflicted methods.
- **OTHER** Other solutions including adopting both contents or creating new code.

IV. RESEARCH TARGETS

In this section, we explain the targets of the survey. We adopted OSS projects which are divided into methods by Historage to survey the history of each method [5].

Historage divides java files into methods and records packages, parameters, and body of 1 method as 1 file for each. Therefore, the differences at method-level can be obtained by the common way at file-level. Kataribe is a hosting service of Historage repositories [4]. TABLE I shows the details of the projects.

TABLE I. DETAILS OF PROJECTS IN KATARIBE

# of projects	conflict	no conflict	error	other
104	10	54	36	4

"no conflict" includes the projects that any conflicts are not detected. "error" and "other" includes the projects whose repository cannot be obtained or part of files cannot be tracked. We surveyed number of merging commits, conflct commits, and conflict methods in available 10 "conflict" projects.

V. RESULTS

TABLE II shows the details of 10 "conflict" projects in TABLE I.

TABLE II. DETAIL OF MERGING AND CONFLICT

project	# of commit	# of merging	# of conflict
james	13581	12	1
jrobin	493	8	3
maven.plugins	39996	109	5
org.eclipse.ajdt	6563	13	1
org.eclipse.gmp.graphiti	2530	48	2
org.eclipse.jubula.core	2746	345	3
org.eclipse.paho.mqtt.java	47	3	2
org.eclipse.scout.sdk	1035	26	3
org.eclipse.stardust.ui.web	3199	12	3
org.eclipse.uml2	2705	38	7

There are a several conflicts in almost 50–40000 commits. We need to note that there can be some hidden conflicts, which cannot be obtained by Git because other version control systems were mainly used before 2011. There are no noticeable correlations between the number of commits, merging, and conflicts.

TABLE III shows the detail of conflict methods in three conflicts of jrobin. Although there are different types of causes, almost all of them are resolved by "1-WAY".

TABLE III. DETAIL OF CONFLICT METHODS OF JROBIN

	# of conflict methods	Causes			Resolutions	
		CHANGE SAME POSITION	DELET-ION	RENAM-ING	1-WAY	OTHER
1st	1	0	1	0	1	0
2nd	364	137	168	59	363	1
3rd	228	31	194	3	228	0
total	593	168	363	62	592	1

TABLE IV shows the detail of conflict methods of all 10 projects.

TABLE IV. DETAIL CONFLICT METHODS OF 10 OSS PROJECTS

project	# of conflict methods	Causes			Resolutions	
		CHANGE SAME POSITION	DELET-ION	RENAM-ING	1-WAY	OTHER
james	115	114	1	0	115	0
jrobin	593	168	363	62	592	1
maven.plugins	34	31	3	0	34	0
org.eclipse.ajdt	2	2	0	0	1	1
org.eclipse. gmp.graphiti	4	2	1	1	3	1
org.eclipse. jubula.core	3	3	0	0	3	0
org.eclipse. paho.mqtt.java	6	0	4	2	6	0
org.eclipse. scout.sdk	7	4	3	0	6	1
org.eclipse. stardust.ui.web	9	9	0	0	9	0
org.eclipse.uml2	6	6	0	0	2	4
total	779	339	375	65	771	8

48% (375/779) of conflict methods are caused by being deleted in one or both parent commits. The ratio of "DELETION" is higher in jrobin, org.eclipse.jubula.core, and org.eclipse.paho.mqtt.java. Conversely, the ratio of "CHANGE SAME POSITION" is higher in james, maven.plugins, org.eclipse.stardust.ui.web, and org.eclipse.uml2. The ratio of "CHANGE SAME POSITION" and "DELETION" is almost equal in other 2 projects, org.eclipse.gmp.graphiti, and org.eclipse.scout.sdk. Few of them were caused by renaming.

99% (771/779) of conflict methods are resolved by adopting the content of one parent method. The ratio of "1-WAY" is very high in all projects excepting org.eclipse.jubula.core and org.eclipse.uml2.

Figure 4 shows an example of "OTHER" in org.eclipse.uml2. Both of methods are conflicted at line 6, and the conflict is resolved by adopting them. Like this, these methods are resolved by combining conflict code—none of them by using new code which does not exist in both.

VI. DISCUSSION

When resolving conflicts, developers usually use merge tools which support resolving them by GUI like kdiff3, or DiffMerge. Most of them visualize the parts where conflicts are detected and display them. Developers fix code using those information. That is the reason why there are no case that new code which are not written in both are not used. However, in the case that conflicts are detected in multiple methods, it is one of the most significant problem that developers must select which method shoud be resolved at first. It is impossible to obtain a satisfactory resolution by only such information [10]. Like that, because existing merging tools don' t give enough information for conflict resolution, it is considered that they have no choice but to adopt one.

The proportion of causes is different for each project. This means that trends of conflicts are affected by characteristics of projects — especially human elements like number of developers who are involed in project or difference of communication shoud be strongly related. We need to analyze some characteristics of projects to find correlative metrics.

In this study, we use difference of files as metrics to detect conflict commits. However, it shoud be important to use not only the number of changed files but also other metrics like the number of changed code lines. It is also important to survey transition of gap in log for each merging commit to verify whether the hypothesis in III-B is correct.

Almost all of the conflict methods are resolved by adopting one method. If some prediction models are structed from data obtained from this study and predict which methods tend to be adopted, it may be possible to extract correlative metrics and contribute supporting for conflict resolution.

VII. CONCLUSION

We surveyed how conflict methods are resolved on a large scale and obtained the proportion of conflict causes and resolutions — almost all of the conflict methods are resolved by adopting one then discarding another.

We plan to survey more projects or consider the validity of our approach, to consider more detailed classification about

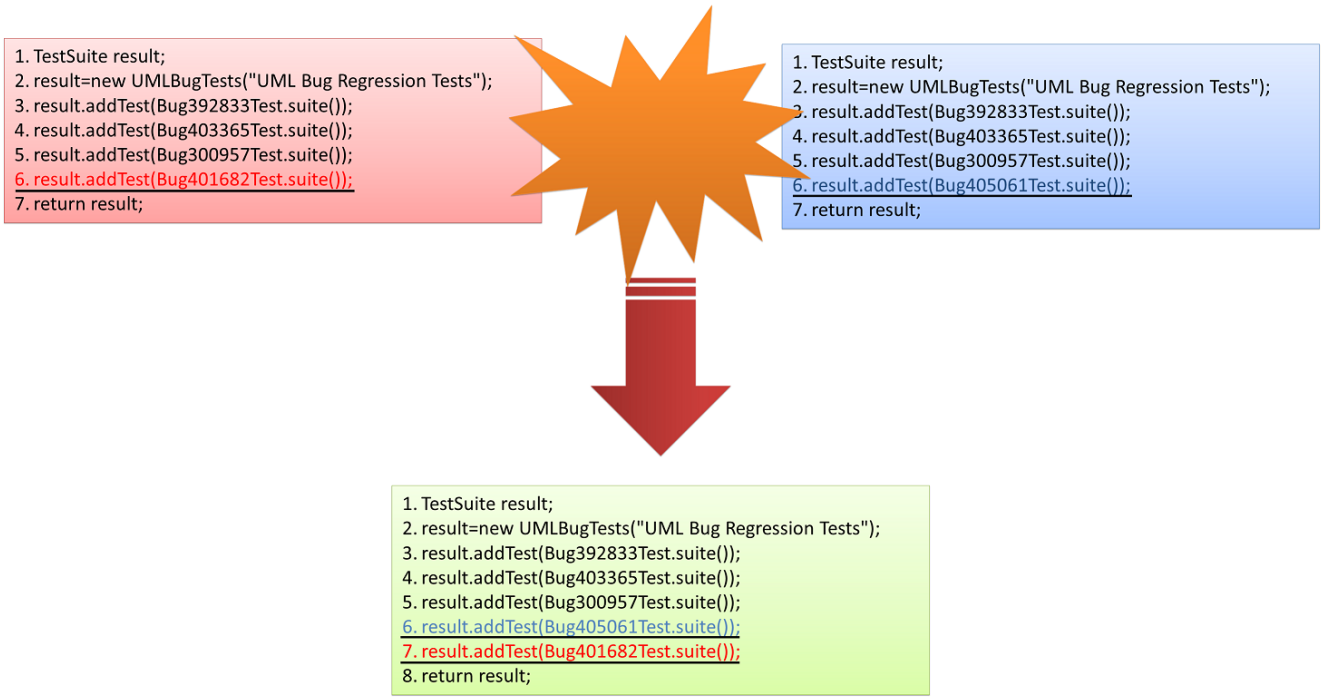

Fig. 4. Example of other: org.eclipse.uml2

resolution, to survey some characteristics of projects to find correlative metrics, and to generate prediction model for support system for conflict resolution.

ACKNOWLEDGMENT

This study has been supported by JSPS KAKENHI Grant Number 26540029, and has been conducted as a part of Program for Advancing Strategic International Networks to Accelerate the Circulation of Talented Researchers.

REFERENCES

[1] S. Apel, C. Kastner, and C. Lengauer. Language-independent and automated software composition: The featurehouse experience. *Software Engineering, IEEE Transactions on*, Vol. 39, No. 1, pp. 63–79, Jan 2013.

[2] Sven Apel, Olaf Leß enich, and Christian Lengauer. Structured merge with auto-tuning: Balancing precision and performance. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pp. 120–129, New York, NY, USA, 2012. ACM.

[3] Sven Apel, Jörg Liebig, Benjamin Brandl, Christian Lengauer, and Christian Kästner. Semistructured merge: Rethinking merge in revision control systems. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pp. 190–200, New York, NY, USA, 2011. ACM.

[4] Kenji Fujiwara, Hideaki Hata, Erina Makihara, Yusuke Fujihara, Naoki Nakayama, Hajimu Iida, and Ken ichi Matsumoto. Kataribe: A hosting service of historage repositories. In *11th Working Conference on Mining Software Repositories (MSR 2014)*, pp. 380–383, 5 2014. Hyderabad, India.

[5] Hideaki Hata, Osamu Mizuno, and Tohru Kikuno. Historage: Fine-grained version control system for java. In *Proc. of 12th International Workshop on Principles on Software Evolution and 7th ERCIM Workshop on Software Evolution (IWPSE-EVOL2011)*, pp. 96–100, 9 2011. Szeged, Hungary.

[6] D. Jackson and D.A. Ladd. Semantic diff: a tool for summarizing the effects of modifications. In *Software Maintenance, 1994. Proceedings., International Conference on*, pp. 243–252, Sep 1994.

[7] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pp. 802–811, Piscataway, NJ, USA, 2013. IEEE Press.

[8] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pp. 3–13, Piscataway, NJ, USA, 2012. IEEE Press.

[9] T. Mens. A state-of-the-art survey on software merging. *IEEE Trans. Softw. Eng.*, Vol. 28, No. 5, pp. 449–462, May 2002.

[10] Nan Niu, Fangbo Yang, J.-R.C. Cheng, and S. Reddivari. A cost-benefit approach to recommending conflict resolution for parallel software development. In *Recommendation Systems for Software Engineering (RSSE), 2012 Third International Workshop on*, pp. 21–25, June 2012.

[11] Shaun Phillips, Jonathan Sillito, and Rob Walker. Branching and merging: An investigation into current version control practices. In *Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering*, CHASE '11, pp. 9–15, New York, NY, USA, 2011. ACM.

[12] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pp. 364–374, Washington, DC, USA, 2009. IEEE Computer Society.