# Supporting Merge Conflict Resolution by Using Fine-Grained Code Change History

Yuichi Nishimura
*Graduate School of Information Science and Engineering*
*Ritsumeikan University, Japan*
*nishimura_y@fse.cs.ritsumei.ac.jp*

Katsuhisa Maruyama
*Department of Computer Science*
*Ritsumeikan University, Japan*
*maru@cs.ritsumei.ac.jp*

***Abstract*—Modern version control systems facilitate concurrent work in software development by providing a mechanism to merge revisions that are independently modified by multiple programmers. However, merge conflicts might emerge due to concurrent modifications, and their resolution might require the programmers to scrutinize every modification in the revisions. This paper presents a tool that can alleviate this cumbersome task of merging conflicting revisions. Our tool exploits the fine-grained edit operation history of Java source code and extracts only the edit operations that affect the revision of a particular class member. By just replaying the extracted edit operations, it helps programmers detect merge conflicts between class members within the two revisions and understand the modifications of the conflicting class members. Moreover, it can artificially merge two snapshots that appear during the evolution of the two revisions and show the programmers a unique artificial snapshot that is consistent with both the merged snapshots. By replaying the fine-grained edits that cause merge conflicts and showing the apparent snapshots with no conflicts as hints of the merged revision, the tool reduces the burden of inspecting the code changes behind the conflicts and reconciling the conflicting revisions.**

***Keywords*-version control; merge conflicts; distributed software development; code changes;**

## I. Introduction

Efficient coordination of multiple programmers is important for effective software development [1]. Since version control systems (e.g., Git and Subversion) support such coordination, their use has become popular in modern software development. Programmers can work on their own workspaces (or repositories) and produce the revisions of their program code in parallel [2]. However, the conflicts among the multiple revisions modified in these independent workspaces might hamper the success of such collaborative development, since different programmers do not know the modifications of others.

Several tools implementing conflict awareness and advanced conflict detection have been proposed to improve traditional version control systems, and their features have been summarized [3]. Whether real-time or event-based (saving or committing), these tools argue that the early detection of merge conflicts can prevent conflict resolutions from escalating. Although this claim is reasonable, do programmers really need additional support? We believe that they not only must know what code fragments (or program elements) are causing merge conflicts but they also have to understand how these code fragments have been changed in the past.

Our approach believes that information about the fine-grained code changes behind merge conflicts are useful for reconciling them. Consider, for example, two code fragments, which are causing a merge conflict, were independently modified by a programmer who will eventually merge the conflicting fragments and her co-worker. If she knows that the changes related to the conflicting code fragments made by her include those made by the co-worker, she might be able to easily or quickly decide on a merging policy; her decision might depend on whether the changes were made at the early or the late stage of his revision work.

This paper proposes a tool called MergeHelper, which exploits the edit operation history of the source code to support the resolution of merge conflicts. In this tool, the edit operations represent fine-grained code changes that are made by a programmer on Eclipse's Java editor. The edit operations are automatically recorded by ChangeTracker [4], which is embedded in MergeHelper. EGit (Eclipse's Git) stores them in a master repository with the changed code when committing the code. Every snapshot of the source code during its evolution can be restored from both the initial (or previous) snapshot and the edit operations that were applied to it.

To detect merge conflicts and uncover the code changes behind them, MergeHelper employs operation history slicing [5], which extracts only the edit operations that affect the revision of a particular class member from the operation history. It identifies the conflicting class members within the two revisions to be merged based on their operation history slices and allows programmers to replay only the edit operations in the slices. In general, the resolution of conflicts requires a deep inspection of the conflicting code fragments and starts with a sufficient understanding of them. Replaying the edit operations related to the conflicts can simplify this time-consuming work. Hattori et. al. demonstrated that the chronological replaying of fine-grained code changes helps programmers find answers to questions related to software evolution [6]. Furthermore, the adoption of operation history slicing is expected to reduce the burden of the inspection work, since the edit operations that are unrelated to the

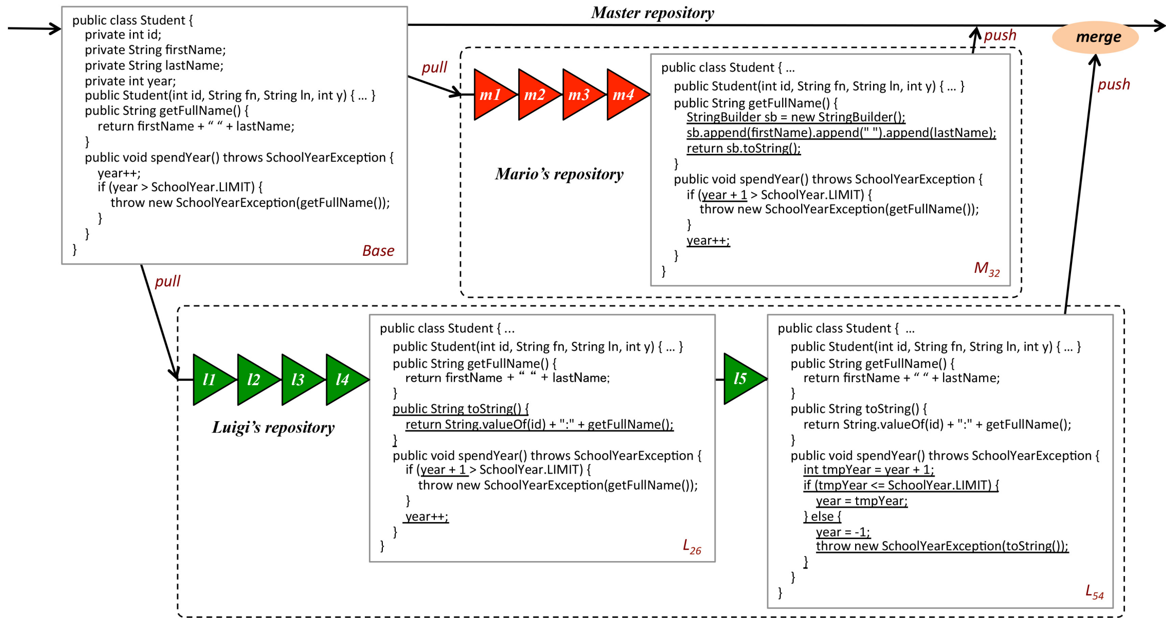


Figure 1. Merging the independently evolved revisions.

conflicts are excluded from the replay.

MergeHelper also employs a mechanism that artificially merges two snapshots appearing during the evolution of the source code in the revisions to be merged. This mechanism allows a programmer to see a unique snapshot that is consistent with both merged snapshots. This apparent snapshot might suggest the resolution of the emerging merge conflicts.

The main contribution of this paper is to present a running implementation of a tool that exploits fine-grained code changes to support the resolution of merge conflicts. This tool also promotes the fusion of conflict detection and program comprehension tools.

## II. Motivating Example

To illustrate the merge conflicts that we tackled, we show an example in Fig. 1. Suppose that Mario and Luigi are co-workers who share the master repository using Git, which stores the base file of the Java source code (simply called Base). Mario and Luige pull (or fetch) changes from the master repository. Both $M_0$ and $L_0$, which are the clones of Base, are stored in their respective local repositories at the beginning. $M_n$ or $L_n$ indicates a snapshot of the source code that is generated immediately after edit operation $n$ $(> 0)$ was applied to its previous snapshot, $M_{n-1}$ or $L_{n-1}$. These snapshots were committed or uncommitted in the local repositories. In Fig. 1, the underlined parts were modified in Mario and Luigi's repositories.

Mario performed these four tasks in the following order:

*m1*) He rewrote the contents of the body of method `getFullName()`.

*m2*) He deleted the entire code at the first line of method `spendYear()`.

*m3*) He inserted "`year++;`" at the last line of `spendYear()`.

*m4*) He modified "`year`" into "`year + 1`" in the conditional expression of the `if`-statement within `spendYear()`.

After the four tasks, consisting of 32 edit operations, Mario successfully pushed the changes ($M_{32}$) in his local repository to the master one without any merge conflicts.

Around the same time, Luigi also performed theses five tasks in the following order:

*l1*) He added a new method `toString()`.

*l2*) He performed the same task as *m4*.

*l3*) He performed the same task as *m3*.

*l4*) He performed the same task as *m2*.

*l5*) He rewrote the contents of the body of `spendYear()` by introducing local variable `tmpYear`.

After the five tasks, consisting of 54 edit operations, Luigi pushed the changes ($L_{54}$) in his local repository to the master one. Unfortunately, this push failed since Mario's changes were already merged in the master repository and are now inconsistent with Luigi's. In the example, both MergeHelper and Git report the conflict that arises from merging the changes with respect to `spendYear()` within class `Student` of $M_{32}$ and $L_{54}$. The changes with respect to `getFullName()` and `toString()` caused no merge conflicts. If Luigi wants a successful push, he has to resolve this conflict by manually editing the conflicting code fragments.

## III. Tool Implementation

MergeHelper only deals with direct conflicts (also known as syntactic conflicts) based on the class members within

parseable Java programs, although several types of merge conflicts [7] emerge in realistic development. Direct conflicts arise when two programmers edit the same code fragment in different ways. MergeHelper detects them between class members (methods or fields) with the same name (the combination of a method or field name and its class name) in the two revisions. Unfortunately, precise detection is impossible if the name (or the signature) of a class member is changed. We ignore indirect conflicts (also known as semantic conflicts) that might cause unwanted behavior or the test failure of the code resulting from the merge.

### A. *Conflict Detection*

To detect merge conflicts, MergeHelper adopts operation history slicing [5]. An operation history slice (or simply a *slice*) is a chronological sequence of the extracted edit operations. The edit operations include manual typing (insertion, deletion, and replacement of text), clipboard editing (copying, cutting, and pasting text), undo/redo actions, and code changes by automatic transformations (code completion, quick fix, formatting, and refactoring).

The detecting procedure consists of the following steps:

1) It collects all the edit operations that were performed during the independent evolution of the two revisions to be merged. Each evolution line is called a *branch*.
2) To find the class members that cause merge conflicts, it selects two class members, $c_1$ and $c_2$, with the same name (not the method signature to cope with signature changes), each of which appears in the last snapshot of each branch, $b_1$ or $b_2$.
3) If not both slices $S(b_1, c_1)$ and $S(b_2, c_2)$ are empty sequences, a merge conflict arises from the code changes of the $c_1$ of $b_1$ and the $c_2$ of $b_2$.

In the example shown in Section II, fifteen slices were extracted from both branches M and L. Four slices are not empty: $S$(M, `getFullName()`), $S$(M, `spendYear()`), $S$(L, `toString()`), and $S$(L, `spendYear()`). As a result, `spendYear()` within $M_{32}$ and $L_{54}$ caused a merge conflict.

### B. *Artificial Merge*

To provide hints for the resolution of merge conflicts, MergeHelper generates a virtual snapshot of the source code by artificially merging two snapshots that appeared during the past evolution of the two revisions. This artificial merge should be performed on demand and restored for every snapshot by the application of each edit operation, which resembles pseudo merges in previous work [1], [8].

The artificially merging procedure consists of the following steps:

1) It selects the first edit operation in each slice $S(b, c)$ for class member $c$ in branch $b$ that causes a merge conflict and finds snapshot $s$ that existed immediately before the application of this edit operation. Then it collects set of snapshots $N(b, c)$, consisting of $s$ and its subsequent snapshots.
2) It selects two snapshots, $s_1$ of branch $b_1$ and $s_2$ of branch $b_2$, from among snapshot sets $N(b_1, c_1)$, which corresponds to $S(b_1, c_1)$, and $N(b_2, c_2)$, which corresponds to $S(b_2, c_2)$. $c_1$ or $c_2$ is the conflicting class member within $s_1$ or $s_2$.
3) If the contents of $c_1$ and $c_2$ are the same, $(s_1, s_2)$ is a candidate pair of the snapshots that will be merged.
4) It checks whether any merge conflicts have arisen for each of the candidate pairs using the conflict detection procedure described in Section III-A. If no conflicts are detected with respect to other class members within snapshots $s_1$ and $s_2$ of $(s_1, s_2)$, these snapshots can be artificially merged.

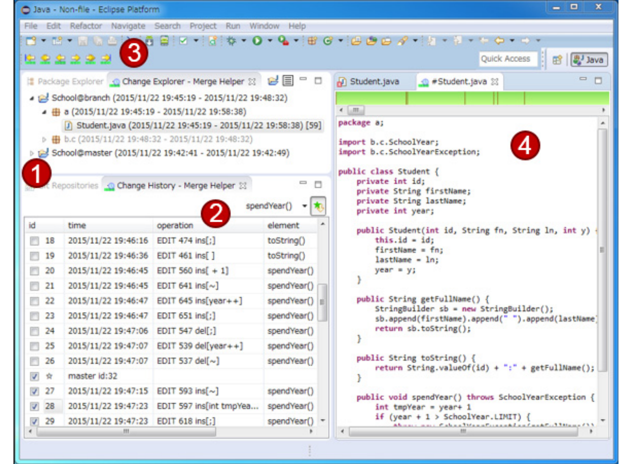


Figure 2. Screenshot of MergeHelper.

In the example shown in Section II, since the contents of `spendYear()` of $M_{32}$ and $L_{26}$ are identical (Fig. 1) by chance, these snapshots cause no conflicts. Thus, $M_{32}$ and $L_{26}$ are artificially merged and the resulting snapshot is presented to the programmer. In realistic development, the opportunities for this artificial merge might be scant, except that the same bug fix is performed.

### C. *Replay Code Changes*

With MergeHelper, a programmer (a tool user) can easily replay the edit operations related to the detected merge conflicts and obtain a virtual snapshot resulting from the artificial merge. Fig. 2 shows a screenshot that consists of four parts: a package explorer ①, an edit operation viewer ②, replay buttons ③, and a source code viewer ④.

The package explorer presents information about the branches and packages containing source files to be merged. The source files that cause merge conflicts are displayed in black and those that are unrelated to the conflicts are displayed in gray. In Fig. 2, the conflicting source file `Student.java` of Luigi's branch is specified.

The edit operation viewer chronologically lists all the edit operations that were performed during the evolution of the specified source file. The resulting snapshots of the artificial merge appear among the edit operations. To see any snapshot that has been restored from the edit operations, a programmer can directly specify the edit operation of interest on this viewer or indirectly specify it by pushing the replay buttons at the top of the package explorer. Moreover, all the edit operations can be selected that are related to one or any combination of the conflicting class members at once using the menu bar located just above this viewer. For example, when a programmer selects `spendYear()` as the conflicting class member on the menu bar, the check boxes of all the edit operations related to this class member are automatically marked. A programmer can also freely mark and unmark the check boxes of the edit operations.

The replay buttons allow programmers to forward and backward replay the edit operations on the list one by one and replay only the marked edit operations. The source code viewer displays snapshots of the source code that exists immediately after the application of the specified edit operation and the resulting source code of the artificial merge.

Programmers can see how the conflicting class members were changed in the past by forward replaying, backward replaying, fast-forwarding, and rewinding the edit operations as if watching an animated movie. This replaying helps answer questions about when the merge conflicts actually arose in the past, what code changes actually caused them, and what code changes were performed before or after their occurrence. Therefore, MergeHelper is useful for resolving merge conflicts.

## IV. Related Work

A number of merge approaches have been proposed [7]. Several tools aim for the early detection of merge conflicts by promoting real-time awareness (e.g., Palantír [9], CollabVS [10], and Syde [11]), by introducing continuous merge (e.g., Crystal [1] and WeCode [8]), and by sharing fine-grained code changes (e.g., CloudStudio [3]). Although these tools do not emphasize the inspection of merge conflicts, MergeHelper especially endeavors to reduce the burden of such inspection by extracting only the fine-grained code changes that might cause conflicts and replaying them.

MergeHelper adopts merge conflict detection based on the hybrid use of the structure of program elements (or class members) and their textual contents (the unstructured aspect). This concept seems to resemble a semi-structured version control tool called FSTMerge [12]. A large difference is that MergeHelper uses fine-grained code changes to inspect the emerging conflicts, but FSTMerge does not explicitly use such code changes to support inspections.

Note that MergeHelper never competes against the conventional tools that detect merge conflicts. We believe that the extension of its implementation is relatively feasible so that it can resolve merge conflicts in cooperation with the conventional tools.

## V. Conclusion

The resolution of merge conflicts is problematic for programmers. Our proposed MergeHelper allows them to efficiently replay the fine-grained code changes related to conflicting class members, which eases their burden. The tool implementation will be available at:
`http://www.fse.cs.ritsumei.ac.jp/mergehelper/`.

MergeHelper's ability should be improved. Unfortunately, its current implementation can only deal with the merge of two branches, since it can automatically neither record the edit operations that achieve the manual merge of two branches nor merge their edit operation histories. In other words, it cannot deal with successive merges of more than two branches. This might be a fatal drawback for realistic development. To overcome this problem, our improved version will employ an additional mechanism based on edit history refactoring [13].

## Acknowledgment

The authors would like to thank Shinpei Hayashi, Takayuki Omori, and Tetsuo Kamina for their valuable comments to our work. This work was partially sponsored by the Grant-in-Aid for Scientific Research (15H02685) from the Japan Society for the Promotion of Science (JSPS).

## References

[1] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Proactive detection of collaboration conflicts," in *Proc. ESEC/FSE '11*, 2011, pp. 168–178.

[2] B. O'Sullivan, "Making sense of revision-control systems," *CACM*, vol. 52, no. 9, pp. 56–62, 2009.

[3] H.-C. Estler, M. Nordio, C. A. Furia, and B. Meyer, "Unifying configuration management with merge conflict detection and awareness systems," in *Proc. ASWEC '13*, 2013, pp. 201–210.

[4] "ChangeTracker," https://github.com/katsuhisamaruyama/changetracker.

[5] K. Maruyama, E. Kitsu, T. Omori, and S. Hayashi, "Slicing and replaying code change history," in *Proc. ASE '12*, 2012, pp. 246–249.

[6] L. Hattori, M. D'Ambros, M. Lanza, and M. Lungu, "Software evolution comprehension: Replay to the rescue," in *Proc. ICPC '11*, 2011, pp. 161–170.

[7] T. Mens, "A state-of-the-art survey on software merging," *IEEE TSE*, vol. 28, no. 5, pp. 449–462, 2002.

[8] M. L. G. aes and A. R. Silva, "Improving early detection of software merge conflicts," in *Proc. ICSE '12*, 2012, pp. 342–352.

[9] A. Sarma, G. Bortis, and A. van der Hoek, "Towards supporting awareness of indirect conflicts across software configuration management workspaces," in *Proc. ASE '07*, 2007, pp. 94–103.

[10] R. Hegde and P. Dewan, "Connecting programming environments to support ad-hoc collaboration," in *Proc. ASE '08*, 2008, pp. 178–187.

[11] L. Hattori and M. Lanza, "Syde: A tool for collaborative software development," in *Proc. ICSE '10*, 2010, pp. 235–238.

[12] S. Apel, J. Liebig, B. Brandl, C. Lengauer, and C. Kästner, "Semistructured merge: Rethinking merge in revision control systems," in *Proc. ESEC/FSE '11*, 2011, pp. 190–200.

[13] S. Hayashi, D. Hoshino, J. Matsuda, M. Saeki, T. Omori, and K. Maruyama, "Historef: A tool for edit history refactoring," in *Proc. SANER '15*, 2015, pp. 469–473.