

コードの編集履歴を用いた競合解決支援

西村 雄一¹ 紙名 哲生^{2,a)} 丸山 勝久^{2,b)}

受付日 2017年8月2日, 採録日 2018年1月15日

概要: ソフトウェアの並行開発において、異なる開発者が同一のソースコードを独立に変更した場合、それらの変更のマージが競合を引き起こすことがある。このような競合を解決するためには、版管理システムに残された過去の変更を細かく調査し、競合の原因やその解決策を見つけ出す必要があるが、これは面倒で時間のかかる作業である。本論文では、開発者がどのように Java ソースコードを編集してきたかという情報を表現した細粒度の編集操作の履歴を用いて、競合解決を支援するツールを提案する。このツールは、競合するクラスメンバに関係する編集操作だけを抽出し、それらを再演することで、競合がどのように発生したのかの理解を助ける。また、過去の編集において競合を発生させずにマージ可能な時点を検出し、自動的にマージを適用した結果を開発者に提示する。簡単な被験者実験を通して、提案ツールが開発者の競合原因の把握や競合解決作業の支援に有用であることを確認した。

キーワード: 並行開発, 版管理, コード変更, 競合解決, プログラム理解

Merge-conflict Resolution Support Using Code Change History

YUICHI NISHIMURA¹ TETSUO KAMINA^{2,a)} KATSUHISA MARUYAMA^{2,b)}

Received: August 2, 2017, Accepted: January 15, 2018

Abstract: In concurrent software development, merge conflicts emerge due to independent modifications that different developers have made. The resolution of such merge conflicts might require the developers to scrutinize every modification in the revisions and thus its task is troublesome and time-consuming. This paper presents a tool that supports merging differences between two conflicting revisions, using the fine-grained edit operation history of Java source code. By both extracting edit operations related to merge conflicts within the two revisions and replaying the extracted edit operations, the tool helps developers understand how such conflicts occurred. Moreover, it can automatically merge two consistent revisions that appear in their respective modifications. This artificially generated code might be a beneficial hint of the successful merge. Through a simple experiment, we confirmed that the tool can reduce the burden of inspecting the code changes behind the conflicts and reconciling the conflicting revisions.

Keywords: concurrent development, version control, code changes, merge conflicts, program comprehension

1. はじめに

ソフトウェア開発の生産性を高めるためには、複数の開発者による効率的な連携が重要である [1]。このため、近

年のソフトウェア開発では、分散共同作業を支援する版管理システム (Git や Mercurial) の利用が一般的となっている。通常、版管理システムを利用した分散開発では、それぞれの開発者が、もとのソースコードが存在するメインブランチから自分独自のブランチ (ワークスペース) を作成し、各自のブランチのソースコードに対して編集を行う。編集が完了したソースコードは、メインブランチにおいてマージ (編集が統合) される。ブランチの作成とマージを適切に実施することで、それぞれの開発者は互いに干渉されずに並行して開発作業を行うことができる [2]。

¹ 立命館大学大学院情報理工学研究科
Graduate School of Information Science and Engineering,
Ritsumeikan University, Kusatsu, Shiga 525–8577, Japan

² 立命館大学情報理工学部
Department of Information Science and Engineering, Ritsumeikan University, Kusatsu, Shiga 525–8577, Japan

a) kamina@acm.org

b) maru@cs.ritsumeikan.ac.jp

一方、このような並行開発では、複数の開発者により独自に編集されたコード片が、もとのソースコードにおいて重なることがあるという問題を根本的に含んでいる。ここでは、それぞれ独立して編集されたコード断片がマージにおいて同一箇所に重なることを、競合 (merge conflict) と呼ぶ。このようなコード断片を1つのソースコードにマージすることは一般的に難しく、これは並行開発の促進を妨げる大きな要因となっている [3]。

通常、競合が発生したソースコードをそのままにしておくことはできないため、競合の解決が必須である。競合の解決とは、同一のコード断片に対する競合する2つの編集のうち、どちらか片方の編集を選択すること、あるいは両方の編集を矛盾することなく混ぜ合わせることを指す。開発者は、自分のソースコードをメインブランチにコミットする際にマージによる競合が発生するかどうかを検査し、競合が発生する場合にはそれを解決する。その際、それぞれのソースコードにおいてどの部分が競合しているのかを正確に把握することが重要であり、この作業には手間と時間がかかる。また、競合に対する理解が不十分なままソースコードのマージ作業を行うと、予期せぬ動作の発生やバグの混入につながる恐れがある。

ここで、Gitのような版管理システムを用いた競合検出では、ソースコードの版間差分に基づく3者間マージ (3-way merge) [4] が主に採用されている。このような3者間マージにおいて発生する競合解決の支援を目的として、マージ手法を改良する研究が従来から行われている。たとえば、ソースコードを単純にテキストとして扱うのではなく、構文解析木で表現されたプログラム構造を考慮したマージ手法 [5], [6] が提唱されている。

しかしながら、ソースコードの差分を活用するマージ手法の改良には限界がある [7]。版管理システムには基本的に各版のソースコードだけが蓄積されている。このため、たとえ文献 [8] の手法によりソースコードの版間差分から編集を抽出したとしても、それらが過去の変更を正確に再現しているわけではない [9]。このような状況において、相手ブランチの2つの変更のうち一方の変更の編集だけを分離して自分のブランチに取り入れることは困難である。たとえば、2つの連続する変更において後の変更における編集が前の変更における編集を上書きしてしまうと、版間差分からは2つの変更を正確に分離することは難しい。Negraらは、変更の37%で、編集の上書きが発生していると報告している [10]。競合の解決を支援するためには、版間差分に残らない細粒度な編集の調査を容易にし、競合に関係する編集だけを分離して開発者の理解を促進するツールが必要である。

そこで、本論文では、開発者が過去に行ったすべての編集操作を集めた履歴を用いることで、並行開発における競合解決を支援するツール MergeHelper を提案する。編集操

作とはソースコード上で実施された細粒度の編集のことを指し、編集対象、編集箇所、編集時刻、編集者、テキストの追加/削除/置換/カット/コピー/ペーストといった編集の内容に関する情報を持つ。また、本論文では、過去のすべての編集操作を集めたものをコード編集履歴と呼ぶ。

MergeHelper は、以下に示す3つの手法により開発者の競合解決を支援する。

- 編集操作やソースコードにおけるクラスメンバ (メソッドとフィールド) の依存関係や対応関係を追跡することで、競合するクラスメンバを開発者の介入なしで自動的に検出する。
- 版間差分には残らない細粒度の編集の調査を効率的に実施できるように、コード編集履歴から競合に関係する編集操作だけを抽出し、それらを再演対象として開発者に提示する。
- 再演対象として提示する編集操作の数を減少させるために、編集操作からソースコードを復元し、競合を発生させずにマージに成功するソースコードを人工的に生成する。

ここで、編集操作を競合の解決の活用するという発想は MergeHelper がはじめてではない。Lippe らは、ソースコードの時間的変化に着目し、コードの変換操作 (本論文における編集操作に相当) に基づくマージ手法を提案している [7]。この手法では、変換操作の順序付け/削除/修正により、コード断片の単純な削除や修正だけでは対処できない競合に対して、自動的なマージの可能性を高めることに成功している。しかしながら、この手法ですべての競合が解決されるわけではなく、さらには開発者が自動的なマージの結果を受け入れるとは限らない。このため、手動により競合を解決しなければならない場面は依然として残り、その支援は必須である。このような観点から、MergeHelper は開発者の競合解決の支援を目指している。また、著者らの知る限り、上記に示す3つの手法をすべて実現した競合解決支援ツールは、MergeHelper のほかにいまだ存在しない。

以降、2章では本論文で扱う競合を説明し、その解決を支援するツール MergeHelper の概要を述べる。3章では MergeHelper の特徴である3つの手法を説明する。4章では MergeHelper の実装と利用例を説明する。5章では MergeHelper を用いた被験者実験の結果を示す。6章で関連研究を紹介し、7章で本論文のまとめと今後の課題を述べる。

2. 編集履歴を用いた競合解決支援ツール

本章では、まずマージの際に発生する競合の例を示す。次に、MergeHelper が再演の対象とする編集操作を説明し、競合の解決に対する MergeHelper の利用シナリオを示す。

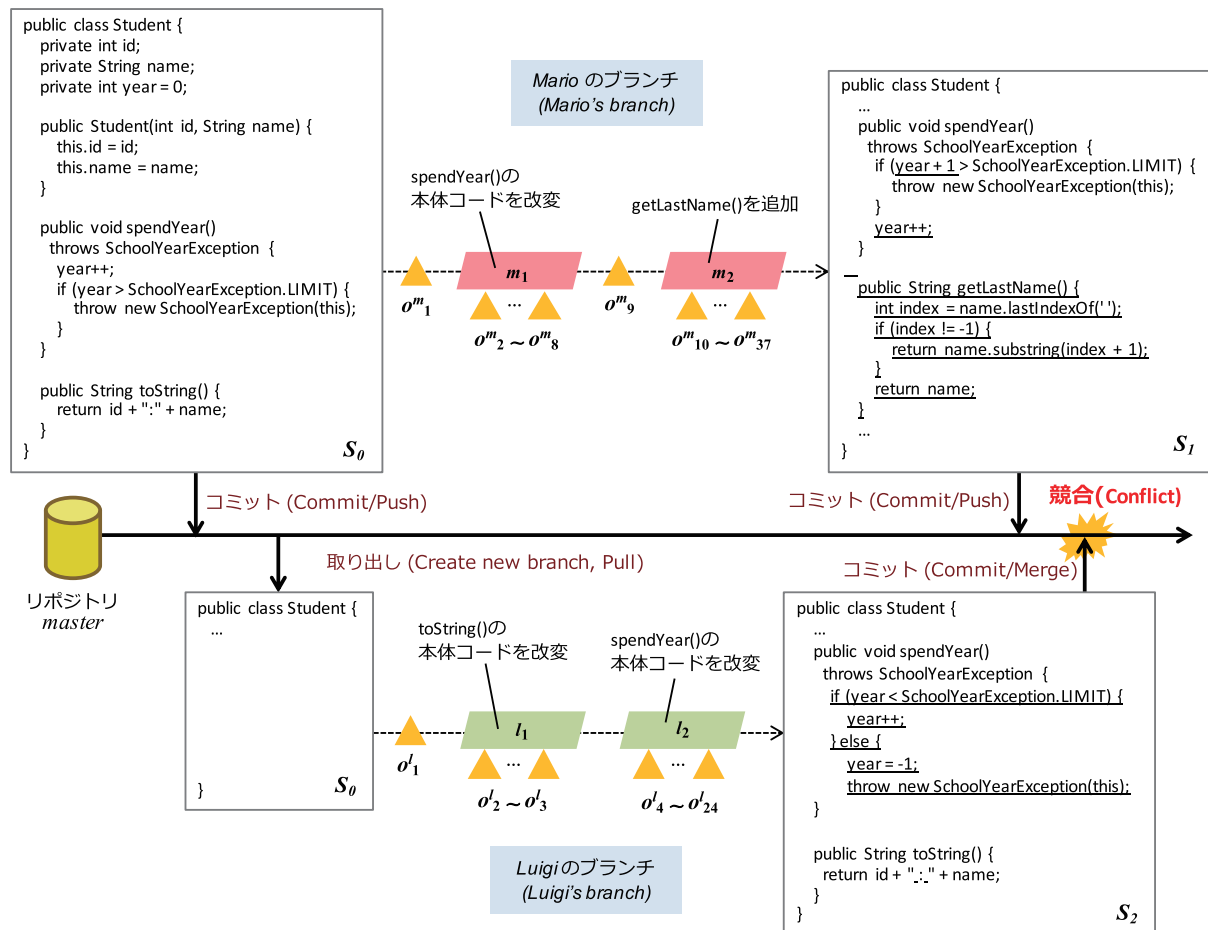


図 1 並行開発における競合の例

Fig. 1 Example of a merge conflict in concurrent software development.

2.1 競合の例

本論文で対象とする競合を説明するための例を図 1 に示す。いま、Git を用いたリポジトリ `master` を共有する開発者 Mario と Luigi がいる。 m_1 と m_2 は Mario が行った変更、 l_1 と l_2 は Luigi が行った変更を指す。 o_1^m や o_1^l は編集操作を指し、2.2 節で後述する。ソースコード中の下線は、それぞれの開発者が編集した箇所を指す。

この例では、まず Mario が Java のソースファイル (`Student.java`) を作成し、そのソースコード S_0 を `master` にコミット (プッシュ) した。さらに、Mario は自分のブランチにおいて開発を続け、 S_0 に対して以下の 2 つの変更を行った。

m_1 : メソッド `spendYear()` の本体コードを改変

m_2 : メソッド `getLastName()` を追加

その後、Mario は完成した `Student.java` のソースコード S_1 を `master` にコミットした。コミット直前の `master` には S_0 が格納されていたため、コミットは問題なく受け入れられ、`master` の最新版が S_1 となる。

Git を利用した開発では、複数の開発者が独自のブランチを作成することで、並行してソースコードの変更を実施することができる。図 1 の例では、Luigi が `master` から

S_0 を取り出す (プルする) ことで、新たなブランチを作成し、そのブランチにおいて以下の 2 つの変更を行った。

l_1 : メソッド `toString()` の本体コードを改変

l_2 : メソッド `spendYear()` の本体コードを改変

その後、Luigi は完成した `Student.java` のソースコード S_2 を `master` にコミットした。しかしながら、コミット直前の `master` には S_0 ではなく S_1 が格納されているため、このコミットはそのままでは成功しない。本論文では、競合によりコミットが失敗した (競合の解決を試みる) 時点を経過したとき、競合解決を試みる S_1 と S_2 を競合解決における変更後ソースコード、 S_1 および S_2 の派生元である S_0 を競合解決における変更前ソースコードと呼ぶ。

このような状況でコミットを成功させるため、Git では 3 者間マージ (以降、Git のマージと呼ぶ) が行われる。このマージでは、独立に編集された 2 つのテキストファイル (図 1 の S_1 と S_2) とそれらの派生元となるテキストファイル (図 1 の S_0) の差分をそれぞれ抽出する。差分が派生元のテキストファイルにおいて同じ箇所に重ならない場合、差分を両方とも含むテキストファイルが自動的に生成される。

残念ながら、図 1 の例では、メソッド `spendYear()` の

```

public void spendYear()
throws SchoolYearException {
<<<<<< HEAD
if (year + 1 > SchoolYearException.LIMIT) {
=====
if (year < SchoolYearException.LIMIT) {
year++;
} else {
year = -1;
>>>>>> refs/heads/luigi
throw new SchoolYearException(this);
}
year++;
}
}

```

図 2 競合の報告

Fig. 2 Report of a merge conflict.

本体コードで差分が重なっているため、図 2 のような競合（一部省略）が報告される。この場合、開発者は手動で競合を解決することになる。たとえば、フィールド変数 `year` の値を `-1` に設定したうえで、例外を送出するには、 m_1 の変更を破棄したうえで、 m_2 , l_1 , l_2 の変更を採用するようにマージすることが考えられる。しかしながら、従来の手動によるマージにおいては S_0 , S_1 , S_2 およびそれらの差分だけしか見ることができず、編集の詳細な情報 (m_1 , m_2 , l_1 , l_2 おける個々の編集操作) まで把握することは困難である。このため、手動マージにおける競合の解決には手間と時間がかかるのが現状である。

2.2 編集操作

MergeHelper では、ソースコードの変更において、開発者が実施した細粒度の編集をその操作で表現し、それを競合解決の支援における再演対象とする。個々の編集操作は、編集対象、編集箇所、編集時刻、編集者、テキストの追加/削除/置換カット/コピー/ペーストといった編集の内容に関する情報を持つ。開発者が実行したカット操作とペースト操作は通常の編集（開発者のタイピング）と区別されたうえで、それぞれテキストの削除と追加をとまなう編集操作として記録される。コピー操作に関しては、コピーされたテキストを保持する編集操作が用意されている。また、ソースコードを直接編集する操作だけでなく、ファイルオープンなどのファイル操作も編集操作に含まれる。

本論文では、ブランチ b の n 番目の編集操作を o_n^b と表す。図 1 の例において、Mario のブランチ m と Luigi のブランチ l で記録された編集操作の数はそれぞれ 37 個と 24 個であった。よって、 m および l における編集操作は、 $o_1^m, o_2^m, \dots, o_{37}^m$ および $o_1^l, o_2^l, \dots, o_{24}^l$ となる。例として o_{24}^m の内容を示すと、「時刻 2017/06/09 12:07:59 において、開発者 Mario が、ファイル Student.java のオフセット 243 の位置にテキスト “+ 1 ” を追加」である。

ここで、図 2 における変更 m_1 および m_2 の編集操作の数はそれぞれ 7 個 ($o_2^m \sim o_8^m$) と 28 個 ($o_{10}^m \sim o_{37}^m$) である（ファイルオープン操作 o_1^m と、メソッド `spendYear()` と `getName()` の間にタブおよび改行を挿入した操作 o_9^m を含

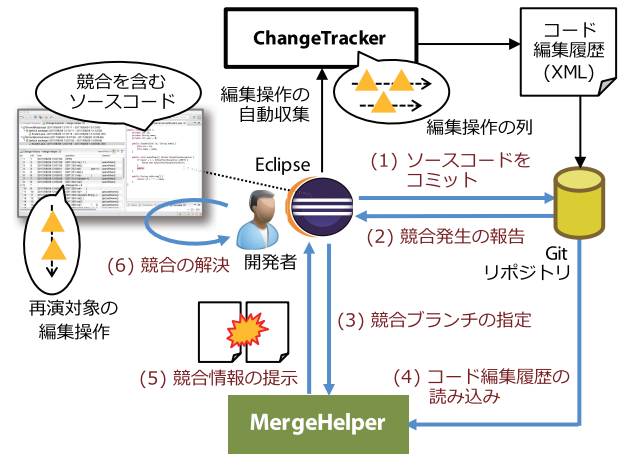


図 3 MergeHelper の利用シナリオ

Fig. 3 Usage scenario of MergeHelper.

わせると合計で 37 個になる）。また、変更 l_1 および l_2 の編集操作の数はそれぞれ 2 個 ($o_2^l \sim o_3^l$) と 21 個 ($o_4^l \sim o_{24}^l$) である（ファイルオープン操作 o_1^l を合わせると合計で 24 個になる）。

2.3 利用シナリオ

MergeHelper は、統合開発環境（IDE: Integrated Development Environment）である Eclipse のプラグインとして実装されている。また、Eclipse から Git 操作ができる EGit^{*1} を利用した Java ソースコードの開発を対象とする。

編集操作の収集と再演を行うために、MergeHelper は ChangeTracker^{*2} を一部拡張して組み込んでいる。ChangeTracker は、開発者が Eclipse の Java エディタ上で実行したすべての編集操作を記録し、それらを XML 形式で出力する。MergeHelper は、XML ファイルに格納されたコード編集履歴を読み込むことで、版管理システムでは取得できない（連続する版間で抜け落ちてしまう）編集を得ることができる。MergeHelper の入力を以下に示す。

- コード編集履歴（収集されたすべての編集操作）
 - 開発者が指定した競合する 2 つのブランチ（の名前）
- また、以下に示す競合情報を出力する。

- 編集が競合するクラスメンバ
- 開発者に再演を促す編集操作の一覧
- 2 つのブランチの途中時点において強制的にマージを実施することで人工的に生成したソースコード

MergeHelper の利用シナリオを図 3 に示す。コード編集履歴は ChangeTracker により自動的に収集される。以下に、開発者と MergeHelper のやりとりを簡単に説明する。

- (1) 開発者が EGit を介して、ソースコードを Git リポジトリにコミットする。
- (2) Git のマージにより競合が検出された場合、それが開

^{*1} <http://www.eclipse.org/egit/>

^{*2} <https://github.com/katsuhisamaruyama/changetracker>

発者に報告される。競合が検出されない場合はマージが成功して終了する。

- (3) Git により報告された競合報告を参考にして、開発者が競合の解決を試みる 2 つのブランチを指定する。
- (4) MergeHelper が、それぞれのブランチにおいて実行されたコード編集履歴を読み込む。
- (5) MergeHelper は競合情報を開発者に提示する。
- (6) 開発者は、提示された競合情報を参考に、編集操作を再演したり、人工的にマージされたソースコードを確認したりすることで競合を解決する。

MergeHelper では、2 つのブランチにおいて、競合解決時の変更後ソースコード（たとえば、図 1 における S_1 と S_2 ）が構文解析可能であることを前提とし、変更後ソースコードに現れるクラスメンバ（メソッドおよびフィールド）を競合の単位として扱う。

ここで、ソースコードをテキスト的にマージ（textual merging）した場合、構文的な競合（syntactic conflicts）と意味的な競合（semantic conflicts）が発生する可能性がある [4]。構文的な競合とは、マージ後のソースコードが構文的に正しくない（構文解析に失敗する）ことを指す。意味的な競合とは、構文的に正しいマージ後のソースコードがコンパイルに失敗したり、開発者の望まない振舞いやテストの失敗を引き起こしたりすることを指す。

3. 競合情報の生成手法

MergeHelper は、3 つの手法（競合の検出、再演対象の抽出、人工的マージ）に基づき、2.3 節で述べた競合情報を出力する。本章では、MergeHelper の特徴であるこれら 3 つの手法を説明する。

3.1 コード編集履歴を用いた競合の検出手法

テキスト的にマージしたソースコードが構文のおよび意味的に正しいことを確認しながら競合を解決するためには、どのプログラム要素が競合に関係しているのかという情報を開発者は知っておいた方が都合がよい。そこで、MergeHelper では、コード編集履歴に蓄積されたそれぞれの編集操作が、どのクラスメンバの本体コードを書き換えたのかを追跡することで、競合するクラスメンバを特定する。ここで、プログラム要素としては、クラスやファイルなどより大きな単位の要素や、文や式などより小さな単位の要素が考えられる。著者らは、競合解決の際に開発者が意識する単位として、Java ソースコードにおいて中心的な役割を果たすクラスの構成要素、かつ名前を持つプログラム要素が適していると考え、クラスメンバを競合検出の単位として扱うことにした。以降、競合するクラスメンバを特定する手順を述べる。

MergeHelper では、開発者が指定したブランチにおいて、特定の期間（もとのブランチにおける一部の区間）に含ま

れる編集操作だけを扱うことがある。ここでは、特定の期間を持つブランチを部分ブランチと呼ぶ。ブランチ b に対して、開始時刻 t_s と終了時刻 t_e で指定された期間と持つ部分ブランチを $b[t_s, t_e]$ と表す。開始時刻と終了時刻を明示的に指定する必要がない場合は $b[]$ と表す。

ブランチ b に含まれるすべての編集操作の集合を $Op(b)$ とすると、 b の部分ブランチ $b[t_s, t_e]$ に含まれるすべての編集操作の集合 $O(b[t_s, t_e])$ は次のようになる。

$$O(b[t_s, t_e]) = \{ o \in Op(b) \mid t_s \leq t_o \leq t_e \}$$

t_o は編集操作 o の実行時刻を指す。

次に、編集操作とクラスメンバの対応関係を定義する。対応関係は、編集履歴スライス（operation history slice）[11], [12] を利用して求める。編集履歴スライスとは、編集操作間の依存関係を追跡することで、特定のクラスメンバの本体コードの記述に影響を与える編集操作だけを、もとの編集履歴から抽出した編集操作の列である。

部分ブランチ $b[]$ の編集履歴から復元可能なすべてのソースコードの集合を $Src(b[])$ 、 $b[]$ に現れるソースコード $s \in Src(b[])$ に存在するすべてのクラスメンバを $Mem(b[], s)$ 、 $b[]$ に現れる s に存在するクラスメンバ $c \in Mem(b[], s)$ の編集操作スライスを $Sq(b[], s, c)$ とおく。条件 $o \in Sq(b[], s, c)$ が成立する場合に、編集操作 o は c と対応関係 $o \leftrightarrow c$ を持つと定義する。これは、 o が c の本体コードを直接、あるいは別の編集編集操作 $o' \in O(b[])$ を介して書き換えている（たとえば、 o でコピーされたテキストが o' で c の本体コードにペーストされている）ことを意味している。

さらに、 $O(b[])$ 内の各編集操作に対応するクラスメンバを集めることで、 $b[]$ において編集されたすべてのクラスメンバの集合 E を次のように定義する。

$$E(b[]) = \{ c \mid o \leftrightarrow c \wedge o \in O(b[]) \}$$

いま、ブランチ $b2$ のソースコードをコミットしたときに、ブランチ $b1$ のソースコードと競合が発生したとする。MergeHelper では、2 つの部分ブランチ $b1[]$ と $b2[]$ に対する $E(b1[])$ と $E(b2[])$ において、同一のクラスメンバ c が存在する場合、マージにおいて c が競合すると判定する。クラスメンバが同一であるとは、それらのシグニチャが同一であることを指す。メソッドのシグニチャは、その名前、引数の数、引数の順序、引数の型、戻り値の型で構成される。フィールドのシグニチャは、その名前と型で構成される。競合するクラスメンバの集合 C は次のようになる。

$$C(b1[], b2[]) = E(b1[]) \cap E(b2[])$$

C が空集合の場合、競合は検出されない。

図 1 の例において、Mario と Luigi のブランチをそれぞれ m と l とする。2.2 節で述べたように、それぞれの部分ブランチ $m[]$ と $l[]$ の編集操作の数は 37 個と 24 個である。よって、編集操作の集合は $O(m[]) = \{ o_1^m, o_2^m, \dots, o_{37}^m \}$ 、 $O(l[]) = \{ o_1^l, o_2^l, \dots, o_{24}^l \}$ となる。

次に、それぞれのクラスメンバの集合を計算すると、 $E(m[]) = \{ \text{spendYear()} \# S, \text{getLastName()} \# S \}$, $E(l[]) = \{ \text{spendYear()} \# S, \text{toString()} \# S \}$ となる。SはクラスStudentを指す。また、 $c \# X$ はクラスXのクラスメンバcを指す。この例では、 $C(m[], l[]) = \{ \text{spendYear()} \# S \}$ となり、競合するメソッドが1つ特定される。

ここで、Gitのマージにおいて発生した競合は、図2のようなテキスト形式で出力される。このテキストを見ると、競合を表現するテキスト（記号<<...と>>...には含まれる部分）がメソッドspendYear()に内包されていることが分かる。よって、編集を追跡しなくても、このメソッドが競合していることを開発者は容易に推測できる。しかしながら、テキスト的な差分からソースコードの変化を正確に追跡することが、つねに容易であるとは限らない[13]。たとえば、図1の例において、MarioとLuigiがレイアウト的に隣り合うメソッドspendYear()とtoString()を変更していた場合、競合を表現するテキストが両方のメソッドにまたがることもある。この場合、テキストの内容や行番号などを頼りに、開発者が競合するクラスメンバを推測することになる。MergeHelperでは、編集操作を利用してクラスメンバの変更を追跡することにより、開発者の介入なしで、競合するクラスメンバを自動的に特定することができる。

また、Guimarãesらの手法では、それぞれのブランチにおける変更前後のソースコードの構文解析結果より、変更に関係するクラスメンバが特定できる場合に限り、競合するクラスメンバを特定することができる[5]。しかしながら、この手法では、本体コードが編集されたメソッドの名前が変更されたり、メソッド抽出により本体コードの一部が新規メソッドに移動したりする場合に、変更に関係するクラスメンバを正確に特定することは難しい。これに対して、MergeHelperでは、競合解決時における変更後ソースコードが構文解析可能であることを前提として編集操作を追跡することで、競合解決における変更前ソースコード（および変更途中のソースコード）が構文解析に失敗しても、変更において編集されたクラスメンバを特定できる。よって、構文解析だけにに基づく従来手法に比べて、競合するクラスメンバが特定できる場面が多い。

3.2 編集履歴スライスを用いた再演対象の抽出手法

MergeHelperは、3.1節の手法で特定したクラスメンバに基づき、競合に関係する編集操作を再演対象として開発者に提示する。その際、競合するクラスメンバと対応関係を持つ編集操作だけでなく、そのクラスメンバの本体コードに影響を与えるすべての編集操作を再演対象とするために、編集履歴スライスを利用する。これにより、競合するクラスメンバの本体コードの一部が他のクラスメンバから複写（コピー & ペースト）および移動（カット & ペー

スト）された場合や、さらには競合するクラスメンバがリファクタリングにより新規に抽出された場合であっても、開発者は編集を追跡することができる。

いま、2つの部分ブランチ**b1**と**b2**で競合が発生し、それぞれの部分ブランチ**b1[]**と**b2[]**で競合するクラスメンバの集合**C[b1[], b2[]]**が特定されている場面を考える。**b1[]**および**b2[]**のどちらか一方の部分ブランチ**b[]**において、競合発生時のソースコードを**s_M ∈ Src(b[])**とすると、**b[]**における再演対象の編集操作の集合**S^b**は次のようになる。

$$S^b(b1[], b2[]) = \{ o \mid o \in Sq(b[], s_M, c) \wedge c \in C[b1[], b2[]] \cap Mem(b[], s_M) \}$$

最終的に、**b[]**における再演対象の編集操作**Q^b(b1[], b2[])**は、**S^b(b1[], b2[])**内のすべての編集操作を時刻順に並べた列と定義する。

図1の例において、MarioとLuigiのそれぞれのブランチ**m**と**l**に対する再演対象の編集操作の列を求めると、**Q^m(m[], l[])** = $\langle o_2^m, o_3^m, \dots, o_8^m \rangle$ と**Q^l(m[], l[])** = $\langle o_4^l, o_5^l, \dots, o_{24}^l \rangle$ となる。この例では、どちらのブランチでもコード断片のコピーやペースト、コード断片の抽出などは実行されていないため、**Q^m(m[], l[])**と**Q^l(m[], l[])**はMarioの実施した変更**m1**における7個の編集操作の集合と、Luigiの実施した変更**l2**における21個の編集操作の集合に一致する。

3.3 コード編集履歴を用いた人工的マージ手法

MergeHelperでは、開発者がリポジトリにコミットしたソースコードだけでなく、その編集操作の履歴を活用できる。そこで、2つのブランチにおいて、編集操作の実行ごとにソースコードを復元することで、競合せずにマージが可能な時点を探査する。たとえば、図1の例において、仮にMarioが**m1**の実施直後にソースコードをリポジトリ**master**にコミットし、Luigiが**l1**の実施直後にソースコードを**master**にコミットした（どちらのコミットが先でもよい）場合、Gitのマージは成功する。つまり、Marioの実施した変更**m1**における編集とLuigiの実施した変更**l1**における編集は競合しなかったはずである。

MergeHelperでは、競合する2つのブランチのソースコードに対して、このような競合せずにマージ可能な時点で開発者がコミットを実行したと仮定し、その時点におけるマージ結果であるソースコードを自動生成する。これを人工的マージ（artificial merge）と呼ぶ。これにより、実際にはリポジトリにコミットされていない人工的なソースコードを開発者は手に入れることができる。

部分ブランチ**b1[]**と**b2[]**に対して、人工的マージを適用する時点は次の2つである。

時点A：**b1[]**と**b2[]**において、MergeHelperが再演対象として提示する編集操作の列をそれぞれ**Q^{b1}(b1[], b2[])**と**Q^{b2}(b1[], b2[])**とおく。各列の中で最も早い時刻を

持つ編集操作 $o_{A1} \in O(b1[])$ と $o_{A2} \in O(b2[])$ の実行直前

時点 B : $b1[]$ と $b2[]$ における編集操作 $o_{B1} \in O(b1[])$ と $o_{B2} \in O(b2[])$ の実行直後で、競合するクラスメンバの集合 $C(b1[], b2[])$ に含まれるすべてのクラスメンバの本体コードが（必然あるいは偶然に）同一の内容、かつ、 $C(b1[], b2[])$ に含まれないすべてのクラスメンバが Git のマージにおいて競合しない時点

一般的には、それぞれの部分ブランチにおいて実行された編集操作の順序や内容によって、人工的マージの適用時点は複数個検出されることがある。これに対して、MergeHelper では、開発者が手動でマージを実施する際に再演対象として提示する編集操作の数をできるだけ削減することを目指し、より後方の適用時点を優先する。このような方針のもと、以下に示す指標 AM を導入する。

$$AM(o_{m1}, o_{m2}) = \#O(b1[t_{m1}, t_{e1}]) / \#O(b2[t_{s1}, t_{e1}]) \\ + \#O(b2[t_{m2}, t_{e2}]) / \#O(b2[t_{s2}, t_{e2}])$$

編集操作 o_{m1} と o_{m2} はそれぞれ $b1[]$ と $b2[]$ において人工的マージが適用される時点の直前の編集操作を指す。また、 t_{m1} と t_{m2} はそれぞれ o_{m1} と o_{m2} の実行時刻を指す。 t_{s1} と t_{e1} はそれぞれ $b1[]$ の開始時刻と終了時刻、 t_{s2} と t_{e2} はそれぞれ $b2[]$ の開始時刻と終了時刻を指す。 $\#O$ は O に含まれる編集操作の数を指す。

図 1 の例を用いて、人工的マージの適用時点を選択する様子を説明する。この例では、3.2 節に示したように、Mario と Luigi のブランチ m と l に対して、再演対象の編集操作の列はそれぞれ $Q^m(m[], l[]) = \langle o_2^m, o_3^m, \dots, o_8^m \rangle$ と $Q^l(m[], l[]) = \langle o_4^l, o_5^l, \dots, o_{24}^l \rangle$ となっている。よって、時点 A は、それぞれの編集操作列 $Q^m(m[], l[])$ と $Q^l(m[], l[])$ の先頭の編集操作 o_2^m と o_4^l の実行直前の時点となる。また、 l と m における編集操作の集合 $O(m[])$ と $O(l[])$ から 1 つずつ編集操作を取り出して時点 B の条件を検査すると、時点 B は o_8^m と o_9^l の実行直後の時点と求まる。

そこで、これら 2 つの時点に対して AM を求めると、 $AM(o_1^m, o_3^l) = 1/37 + 3/24 \approx 0.152$ 、 $AM(o_8^m, o_9^l) = 8/37 + 9/24 \approx 0.591$ となる。 $AM(o_1^m, o_3^l) < AM(o_8^m, o_9^l)$ であるため、 o_8^m と o_9^l の実行直後が人工的マージの適用時点として優先的に選択される。

編集操作の実行順序によっては、人工的マージの適用時点が部分ブランチの初期にしか検出されないことがある。この場合、開発者が再演する編集操作の削減の効果がそれほど期待できない。そこで、MergeHelper では、人工的マージにおいて、その適用時点をより後方に移動させるために、編集履歴において編集操作の並べ替え [14], [15] を採用している。並べ替えの単位は、同一のクラスメンバと対応関係を持つ連続する編集操作の列である。並べ替えは、競合するクラスメンバに対応する編集操作列の位置を、そのクラスメンバに対応しない編集操作列の位置より後方に

移動させることで実施する。

いま、2 つの連続する編集操作の列 U_A と U_B を考える。もし U_A 中の任意の編集操作 o_A が U_B 中の任意の編集操作 o_B に依存する (o_A と o_B の順序が入れ替えられない) 場合、 U_A と U_B は並べ替え (スワップ) できない。反対に、 U_A のすべての編集操作が U_B のどの編集操作にも依存しない場合、 U_A と U_B は並べ替え可能である。

図 1 の例に示す Mario のブランチ m において、編集操作の列 $\langle o_2^m, o_3^m, \dots, o_8^m \rangle$ がメソッド `spendYear()`、 $\langle o_{10}^m, o_{11}^m, \dots, o_{37}^m \rangle$ がメソッド `getLastName()` に対応する。 o_1^m と o_9^m は対応するクラスメンバが存在しない。よって、 $m[]$ から 4 つの編集操作の列 $U_1 = \langle o_1^m \rangle$ 、 $U_2 = \langle o_2^m, o_3^m, \dots, o_8^m \rangle$ 、 $U_3 = \langle o_9^m \rangle$ 、 $U_4 = \langle o_{10}^m, o_{11}^m, \dots, o_{37}^m \rangle$ が生成され、その順序は $U_1 \rightarrow U_2 \rightarrow U_3 \rightarrow U_4$ である。この例では、 U_2 が競合するメソッド `spendYear()` に対応する。そこで、 U_2 をこのメソッドに対応しない U_3 および U_4 より後方に移動させるように並べ替えが実施され、最終的な順序は $U_1 \rightarrow U_3 \rightarrow U_4 \rightarrow U_2$ となる。

開発者が細かな変更ごとにソースコードをコミットしていれば、人工的マージが適用できる機会は減少する。また、変更がもつれないようにコミットが実施されていれば、Git の `blame` コマンドを利用して競合に関係する過去のコミットを特定し、MergeHelper の人工的マージに相当する作業を手動で実行することはそれほど難しくない。一方で、開発者が細かな変更ごとにコミットを実行する保証はない。たとえば、Herzig らが調査したオープンソースプロジェクトでは、バグ修正のうち 7%~20% が他の変更と混ざっていることが報告されている [16]。このような変更がもつれた状況でも、競合を含まないソースコードを人工的マージの結果として開発者に提示できる可能性があることが MergeHelper の利点である。

4. ツールの実装と利用例

本章では、MergeHelper の実装について説明し、競合解決支援における利用例を示す。

4.1 実装モジュール

MergeHelper は、次に示す 7 つのモジュールで構成されている。

- (1) コード編集履歴の収集
- (2) 部分ブランチの抽出
- (3) 編集操作とクラスメンバとの対応関係の特定
- (4) 競合するクラスメンバの特定
- (5) 再演対象となる編集操作の抽出
- (6) 人工的マージの適用
- (7) 編集操作列の並べ替え

以下に、それぞれのモジュールを説明する。

4.1.1 コード編集履歴の収集

MergeHelper は ChangeTracker の出力であるコード編集履歴を利用する。ChangeTracker は、Java エディタ上のファイルが保存されたり、閉じられたり、リファクタリングされたりするたびに、メモリに蓄積していた編集操作を XML 形式の編集履歴ファイルに書き出す。

このモジュールは ChangeTracker の出力を常時監視し、編集履歴ファイルが出力された際、そのファイルのコピーを Git 管理下のディレクトリ（フォルダ）に作成する。開発者が編集履歴ファイルをコミットの対象に指定することで、ソースコードをコミットした際に編集履歴も同時に Git リポジトリに保存される。

4.1.2 部分ブランチの抽出

開発者が競合する 2 つのブランチを指定した際、競合に関係する編集操作を含む部分ブランチを抽出する。いま、競合する 2 つのソースコードを S_X と S_Y 、それらの派生元ソースコードを S_B とおく。部分ブランチの開始時点は、 S_B をコミットした時刻、あるいは S_B をリポジトリから取り出すことで新たにブランチを作成した時刻を指す。部分ブランチの終了時点は、 S_X あるいは S_Y をコミットした時刻を指す。

このモジュールは Git コマンドの実行を捕捉し、ブランチに対するコミット操作（プッシュ、プル、マージ）の情報を取得する。Git コマンドの捕捉には、JGit^{*3}の API を利用している。取得する情報は、操作の実行時刻、操作に関係するコミットの識別子やそれらの親コミットの識別子、操作実行時のソースコードのスナップショットである。

4.1.3 編集操作とクラスメンバとの対応関係の特定

3.1 節で述べた競合の検出手法において、競合するクラスメンバを特定するためには、編集操作とクラスメンバの対応が事前に特定されている必要がある。そのために、このモジュールでは、ChangeTracker により提供されている編集履歴スライス抽出モジュールを利用する。ただし、ChangeTracker のモジュールでは、編集履歴スライスの精度を向上させるため、文献 [12] で提唱されている編集履歴グラフをそのまま利用していない。実際には、編集操作間の依存関係を文献 [17] で提唱されている編集操作の交換可能性により定義し、編集履歴グラフを作成したうえで、編集履歴スライスを抽出している。

ここで、編集履歴スライスを利用しなくても、編集履歴から復元したソースコードに対して構文解析を適用し、クラスメンバの存在範囲と編集位置を単純に比較することで、編集操作とクラスメンバの対応関係を推測する方法も考えられる。しかしながら、実際のソースコード編集においては、(1) 復元したソースコードが構文解析できない場合、(2) 名前変更によりメソッドやフィールド変数の追跡が途

切れる場合、(3) カット/コピー/ペースト操作によりコード断片の移動や複製が行われる場合があり、編集位置を単純に比較する方法では対応関係の推測の正確さが低下する。一方、編集履歴スライスを求める際には、各編集操作に対する編集位置を調整するための計算が必要であり、ブランチに大量の編集操作が存在する場合には対応関係の特定コスト（時間とメモリ）が大きくなる。そこで、MergeHelper では、編集履歴スライスを利用する方法と編集位置を単純に比較する方法を開発者が選択できるようにしている。

4.1.4 競合するクラスメンバの特定

このモジュールでは、3.1 節で述べた競合の検出手法を実装したものである。編集操作の集合、クラスメンバの集合、編集操作とクラスメンバとの対応関係を受け取り、競合するクラスメンバを特定する。

4.1.5 再演対象となる編集操作の抽出

このモジュールは、3.2 節で述べた再演対象の抽出手法を実装したものである。編集操作の集合、クラスメンバの集合、競合するクラスメンバの集合を受け取り、MergeHelper が再演対象として開発者に提示する編集操作を抽出する。

ここで、開発者が競合の解決を試みる場合において、再演対象として提示された編集操作だけでなく、ブランチ内のすべての編集操作を再演したいと考えることが想定される。そのため、MergeHelper では、このモジュールで抽出された再演対象の編集操作と区別して、それぞれの部分ブランチに含まれるすべての編集操作を再演できるようになっている。

4.1.6 人工的マージの適用

このモジュールは、3.3 節で述べた人工的マージ手法を実装したものである。再演対象およびすべての編集操作の集合を受け取り、擬似的なコミットを実行し、開発者に提示するソースコードを生成する。

実際に人工的マージを行う際には、既存のリポジトリに影響を与えないように別途リポジトリを作成し、人工的マージの適用により生成したソースコードのコミットを試みる。残念ながら、編集操作から復元されるすべてのソースコードが必ずしも構文解析可能であるとは限らない。このため、クラスメンバが正しく特定できないことがあり、3.2 節で述べた手法で抽出される再演対象の操作列から、競合に関係のある編集操作が漏れることがある。それにより、時点 A において復元した 2 つのソースコードが、Git のマージに失敗することがある。そこで、このモジュールでは、JGit の API を利用して、実際に復元したソースコードに対してマージを適用し、マージに成功した場合に生成されたソースコードを人工的マージの結果としている。マージに失敗した場合は、次に AM の値が大きくなる人工的マージの適用時点を選択し、マージを繰り返し適用する。すべての実行時点で失敗した場合には、人工的マージの結果を開発者に提示しない。

^{*3} <https://eclipse.org/jgit/>

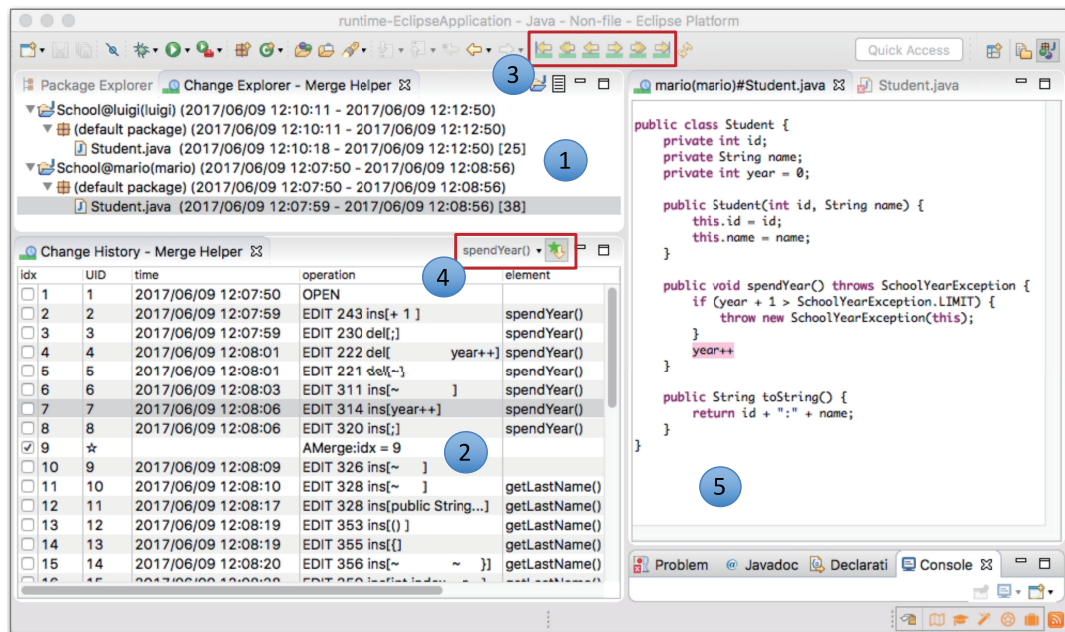


図 4 MergeHelper における競合解決支援の様子

Fig. 4 Screenshot of MergeHelper supporting conflict resolution.

4.1.7 編集操作列の並べ替え

このモジュールでは、3.3 節で述べた人工的マージ手法において、もとの編集操作の列に並べ替えを実施する。実際に編集操作の列を並べ替える際には、並べ替えられた各編集操作に対して、編集箇所を指すオフセット値の調整 [14] を実施する。

4.2 ツールの利用例

図 3 の利用シナリオに示したように、競合が発生した場合、開発者がその解決を試みる 2 つの競合ブランチを指定する。その際、(a) 編集操作とクラスメンバの対応付けに編集履歴スライスを利用するかどうか、(b) 編集操作列の並べ替えを適用するかどうか、(c) 人工的マージを適用するかどうかについても同時に指定する。

MergeHelper から開発者に競合情報が提示されると、図 4 のようなパースペクティブが Eclipse 上に現れ、編集操作の再演が可能となる。それぞれのビューの位置やサイズは変更可能である。また、MergeHelper のビューやボタンは、ChangeTracker の機能を一部改変して実現している。以下、①、②、⑤のビューに関して詳しく説明する。

①の「Change Explorer」ビューにおいて、開発者は、競合の解決を試みる 2 つの部分ブランチや、それらに含まれる Java ソースファイルを指定することができる。指定されたソースファイルの編集操作の一覧が②のビュー、復元されたソースコードの内容が⑤のビューに表示される。パッケージ名やファイル名の後ろには、それらに関する編集履歴の時刻（期間）情報と編集操作の総数（人工的マージの時点を目指す操作も含む）が表示される。

②の「Change History」ビューでは、それぞれの編集操作の情報をしたり、特定の編集操作を選択したりすることができる。編集操作列の並べ替えが適用されている場合は、その結果が反映された編集操作の一覧が表示される。図 4 は、開発者が①のビューにおいて部分ブランチ *mario* の *Student.java* ファイルを選択したときの編集履歴が表示されている。ビュー上に見える UID=1 から UID=14 の編集操作が、2.2 節で示した編集操作 $o_1^m \sim o_{14}^m$ に該当する。編集操作の一覧において、UID 列は各編集操作に割り振られた識別番号を指す。time 列には編集操作の実行時刻、operation 列には編集操作の種類とその簡単な内容が表示される。また、element 列には、各編集操作に対応するクラスメンバが表示される。対応するクラスメンバが存在しない場合は空欄となる。ここで、idx=9 の編集操作を見ると、その種類は“AMerge”と表示されていることに気付く。これは、人工的マージの適用時点を指しており、この編集操作を選択することで人工的マージにより生成されたソースコードを⑤のビューで閲覧することができる。

②のビューにおいて編集操作を直接選択することで、その編集操作の実行直後のソースコードを⑤のビュー上で閲覧することができる。また、キーボードの上下矢印キーや③のアイコンをクリックすることでも編集操作の再演は可能である。さらに、④のアイコンをクリックすることで、表示される編集操作のフィルタリングを切り替えることができる。たとえば、特定のクラスメンバに対応付けされている編集操作のみを表示したり、競合に関する編集操作のみを表示したり、人工的マージの適用時点より後の編集操作のみを一括して再演対象としたりすることができる。

⑤のビューでは、編集履歴から復元したソースコードや人工的マージにより生成されたソースコードを閲覧することができる。図4には、開発者が②においてUID=7の編集操作を選択したときのソースコードが表示されている。

5. 実験

競合の理解支援や解決支援に対する MergeHelper の有用性を確認するために、被験者実験を実施した。本章では、まず実験の準備と手順を 5.1 節と 5.2 節で説明する。その後、実験におけるソースコード、再演対象の編集操作数、被験者へのインタビュー結果を 5.3 節、5.4 節、5.5 節に示す。最後に、インタビューから得られた知見と妥当性の脅威を 5.6 節と 5.7 節で述べる。

5.1 実験準備

MergeHelper の評価実験を行うためには、競合が発生した Java プロジェクトが必要である。残念ながら、過去に競合が発生しており、かつ、その競合に関係する編集履歴がブランチごとに蓄積されているプロジェクトは見つけられなかった。そこで、本実験では、競合に関係する編集履歴を次のように用意した。

- (1) OperationRecorder [18] を用いて過去に実施された被験者実験において、開発者 R が Java Swing による GUI アプリケーションを作成中に収集したコード編集履歴を用意した^{*4}。開発者 R は、2012 年当時、計算機科学を履修していた修士学生であり、MergeHelper が存在することを知らない。
- (2) 編集操作の収集漏れや収集の重複により編集操作のオフセット値がずれることで、挿入あるいは削除テキストに不整合が発生し、ソースコードの復元に失敗することがある。そこで、本実験の実施に先立ち、ソースコードの復元が正しく行えるように、編集操作の収集間違いを著者らが手動で修正した^{*5}。修正後の編集操作の数は 3016 である。ここでは、 i 番目の編集操作を o_i と表す。
- (3) 被験者用のブランチ r を作成し、 o_{2449} の実行直後に復元したソースコード S_{2449} を r にコピーする。その後、 S_{2449} を実験用の *master* ブランチにコミットし

^{*4} MergeHelper に組み込まれている ChangeTracker は OperationRecorder で収集された編集履歴を読み込むことができる。

^{*5} OperationRecorder では、編集操作の収集間違いに備えて、ファイル操作やエディタの活性化などのタイミングでソースコードのスナップショットを記録している。復元に失敗した場合、記録したスナップショットから収集に漏れた編集を推測し、編集操作の追加や直前の編集テキストへの挿入で対応した。一方、編集操作の重複によりソースコードの復元に失敗した場合、余分な編集操作を取り除いた。これらの修正は、他の編集操作の編集内容を取り消さないように実施されており、実験結果に大きな影響を与えることはないと考えている。ちなみに、MergeHelper に組み込まれている ChangeTracker では、編集操作の記録時に自動的に修正が実施されるので、このような手動による修正は不要である。

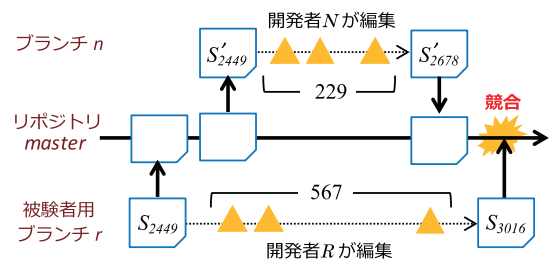


図5 本実験における競合

Fig. 5 Merge conflict in the experiment.

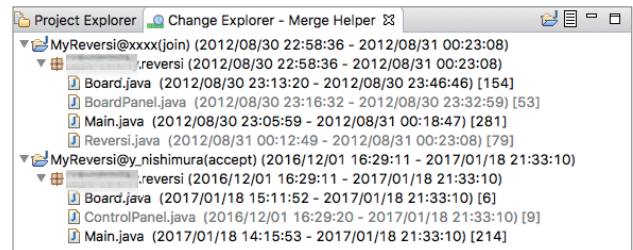


図6 競合するソースファイル情報

Fig. 6 Information on conflicting source files.

た。さらに、開発者 R の編集履歴における o_{2450} から o_{3016} までの編集操作を抜き出し、これらを r 上で行われた編集と見なす。これは、該当する編集操作を記録した XML ファイルを r の管理下にコピーすることで実現する。

- (4) 開発者 N (本論文の第1著者) のブランチ n を作成し、 S'_{2449} ($= S_{2449}$) を *master* から取り出す。その後、開発者 N は n において、開発者 R の開発と独立に S'_{2449} を編集した。この編集は意図的に競合を発生させるように行われており、編集操作の数は 229 である。その後、開発者 N は編集後のソースコード S'_{2678} を *master* にコミットした。

この状況で、被験者が S_{3016} を *master* にコミットすると競合が発生する。本実験における競合の発生状況を図5にまとめる。Git により報告される競合はソースファイル *Main.java* である。

競合の発生を受けて、MergeHelper を起動したときの「Change Explorer」ビューを図6に示す。MyReversi プロジェクトの *y_nishimura* ブランチ (本論文ではブランチ n と呼んでいる) と *xxxx* ブランチ (本論文ではブランチ r と呼んでいる) において、2つのソースファイル *Board.java* と *Main.java* で競合が発生しているのが分かる (競合が発生していないファイルはその名前が灰色で表示される)。Board.java については、Git のマージが成功しているが、MergeHelper では同じメソッドがそれぞれのブランチで編集されているため競合と報告される。

ここで、今回の実験において、 S_{2449} を部分ブランチの開始時点とした理由を述べる。実験では、被験者は開発者 R の立場になり競合解決を試みる。このため、編集内容が

比較的理解しやすいことが重要であると考えた。開発者 R は、 o_{2450} 以降の編集において GUI アプリケーションのメニューバーの実装を主に `Main.java` のクラス `Main` に対して行っていた。最終的には (`Main.java` に関する 281 個の編集操作のうち 262 番目の編集操作で)、開発者 R は新規にソースファイル `Reversi.java` を作成し、メニューバーの実装コードをそのソースファイルのクラス `Reversi` に移動している。これらの編集内容は GUI の見た目の変化に直感的に結び付いている。たとえば、メニュー項目を追加するコードを記述すると、メニュー項目が追加されたことを GUI の見た目で確認できる。よって、被験者が理解しやすいと判断した。

開発者 N は、`Main` のメソッド `main()` の肥大化を防ぐため、`EXTRACTMETHOD` リファクタリングを手動で適用し、新規メソッド `addMenuItems()` を抽出している。さらに、`Main` において、2つのフィールド変数 (`Borad` 型と `JFrame` 型) を追加している。これらのフィールド変数の追加にともない、それを参照するコード断片を `main()` に記述したため、競合が発生している。この `main()` に関する競合は `Git` の報告に含まれるため、それを見つけることは被験者にとって比較的容易である。

これに対して、`EXTRACTMETHOD` により新たに抽出された `addMenuItems()` は `Git` による競合の報告から漏れている。このため、`master` に蓄積されている S_{2449} ($= S'_{2449}$) と S'_{2678} 、および被験者がコミットを試みた S_{3016} を見るだけで、`EXTRACTMETHOD` の適用に気付くことは難しい。さらに、被験者は、このリファクタリングの適用に気付いた後、開発者 N が `addMenuItems()` に対して実施した変更と開発者 N が `main()` に対して実施した変更の両方を理解する必要がある。よって、競合の解決において、このリファクタリングにともなう変更を矛盾なく取り入れることは面倒な作業となることが予想される。

また、本実験において、開発者 N は、開発者 R が編集中の `Board.java` のクラス `Board` のメソッド `restart()` に対して、フィールド変数への代入文 (1 行) を追加している。この変更は開発者 R がコミットする S_{3016} において、`restart()` にそのまま取り入れても競合は発生しない。よって、この競合を被験者が解決することはきわめて容易である。実際に、被験者は `Git` のマージ結果をそのまま受け入れている。

5.2 実験手順

競合解決の作業を実施する被験者は、2017 年 1 月当時、計算機科学を履修していた修士学生 2 名である (被験者 K と T と呼ぶ)。この 2 名は、日頃から `Eclipse` を利用して `Java` プログラムを開発しており、`Git` を含めた版管理システムに関する知識を持っている。`MergeHelper` を用いた実験手順は、次のようになる。

- (1) 開発者 R が開発していた GUI アプリケーションの概要およびソースコード S_{2449} の動作について、著者の 1 人が被験者に説明する。その際、競合する 2 つのブランチ n と r において、主にメニューバーに関する編集が行われていることも伝えている。
- (2) 被験者が `MergeHelper` の操作に慣れるために、本論文の第 1 著者が 2.3 節の利用シナリオに基づき、`MergeHelper` の使用方法や機能を被験者に説明する。その後、開発者 R の編集履歴を `MergeHelper` に読み込ませ、 o_{2450} 以降に実行された編集操作を再演することで、開発者 R が S_{2449} に行った編集を被験者に理解してもらう。ここでは、 R における編集操作の再演のみを許可し、`MergeHelper` の出力する競合情報は被験者に提供しない。
- (3) 被験者は、`MergeHelper` が出力する競合情報を確認したり、競合する 2 つのブランチ n および r の編集操作を `MergeHelper` で自由に再演したりすることで、競合の解決を試みる。このとき、再演の回数やオプション選択などについて、特に制限していない。
- (4) 被験者は、自身が競合を解決したと判断した後、インタビューにおいて質問に答える。

本実験では、競合解決後の GUI アプリケーションのメニューバーに関して、何らかの実装 (コードの改変) が行われており、そのソースコードがコンパイル可能かつ動作する (コンパイル後のプログラムが実行時例外で停止しない) とき競合が解決した条件と見なし、この条件を被験者に伝えている。ただし、マージ結果の正解は 1 つでないため、被験者による判断が適切であった保証はない。

5.3 実験におけるソースコード

本実験において準備したソースファイルと被験者が作成したソースコードの行数を表 1 に示す。図 6 に示したように、本実験において、ブランチ n と r で編集されたソースファイルは全部で 5 つである。 S_{2449} 、 S'_{2449} 、 S'_{2678} 、 S_{3016} は、図 5 に現れるソースコードを指す。また、括弧内の数字は、それぞれのソースファイルにおいて追加された行数と削除された行数 (diff により検出された行差分) を表す。たとえば、「 S_{3016} ($\leftarrow S_{2449}$)」の列の `Board.java` における「+45-5」は、 S_{2449} と S_{3016} の `Board.java` において 45 行の追加と 5 行の削除が検出されていることを表している。「+0-0」は、そのソースファイルが編集されていないことを表す。また、「-」はソースファイルが存在しないことを表す。 S_{git} は `Git` のマージが出力したソースコードを指す。この実験では、`Main.java` だけが `Git` のマージに失敗し、このソースファイルには 26 行の競合の報告が含まれている。 S_K と S_T は、競合の解決において、被験者 K と T が最終的に作成したソースコードを指す。

ここで、 S_K の行差分を見ると、被験者 K は自分のブラン

表 1 本実験において作成されたソースコードの行数

Table 1 Numbers of lines of code for each source file in the experiment.

ソースファイル	S_{2449} (S'_{2449})	S'_{2678} ($\leftarrow S'_{2449}$)	S_{3016} ($\leftarrow S_{2449}$)	S_{git}	S_K ($\leftarrow S_{git}$)	S_T ($\leftarrow S_{git}$)
Board.java	260	262 (+2-0)	300 (+45-5)	301 (MH)	301 (+0-0)	301 (+0-0)
BoardPanel.java	110	110 (+0-0)	122 (+15-3)	122 (S_{3016})	122 (+0-0)	122 (+0-0)
ControlPanel.java	52	40 (+3-15)	52 (+0-0)	40 (S'_{2678})	39 (+0-1)	40 (+0-0)
Main.java	23	82 (+62-3)	17 (+4-10)	88 (MH & G)	25 (+1-64)	27 (+12-73)
Reversi.java	–	–	85 (+85-0)	85 (S_{3016})	85 (+0-0)	91 (+75-69)
Total	445	494 (+67-18)	576 (+149-18)	–	572 (+1-65)	581 (+87-142)

*MH: MergeHelper により検出された競合, G: Git のマージにより検出された競合

表 2 本実験において MergeHelper が提示する再演対象の編集操作の数

Table 2 Numbers of edit operations that MergeHelper presented in the experiment.

ソースファイル	Board.java						Main.java					
ブランチ	n		r		Total		n		r		Total	
クラスメンバ	cc ₁	All ₁	cc ₁	All ₁	cc ₁	All ₁	cc ₂	All ₂	cc ₂	All ₂	cc ₂	All ₂
Original	5	6	8	154	13	160	124	214	261	281	385	495
AMerge	5	5	8	154	13	159	96	177	229	244	325	421
AMerge+Swap	5	5	8	8	13	13	96	96	229	229	325	325

チ r だけに存在する (開発者 R が作成した) Reversi.java の編集を優先して取り入れ, Main.java の競合を解決しているように見える. 一方, S_T の行差分を見ると, 被験者 T は Main.java と Reversi.java の両方を編集することで, 競合の解決を試みているように見える. また, 被験者 K と T が最終的に作成したソースファイルの行数はほぼ等しい (572 行と 581 行) にもかかわらず, 被験者 T の行差分の総数 (1 行+65 行) は被験者 K の行差分の総数 (87 行+142 行) に比べて 3 倍以上となっている. 競合の解決において, 実験手順 (3) に費やした時間は, 被験者 K が約 8 分, 被験者 T は約 22 分であった.

5.4 再演対象の編集操作数

被験者に対するインタビュー結果を述べる前に, 実験手順 (3) において MergeHelper が再演対象として提示する編集操作の数を表 2 に示す. 「Original」の行は, 人工的マージと編集操作の並べ替えを適用しない場合の数を指している. 「AMerge」の行は, 人工的マージが成功した時点より前の編集操作を除外した数を指している. また, 「AMerge+Swap」の行は, 編集操作の並べ替えと人工的マージを適用し, 人工的マージに成功した時点より前の編集操作を除外した数を指している. cc₁ の列は Board.java において競合するメソッド restart(), cc₂ の列は Main.java において競合するメソッド main() に関する編集操作の数を表す. 「All₁」および「All₂」の列は, それぞれ Board.java および Main.java に存在するすべてのクラスメンバに関する編集操作の数を表す.

いま, 開発者が構文的な競合の解決を試みている場合を考えると, 競合するクラスメンバを変更する編集操作だけ

を再演するのが一般的である. その際, もし MergeHelper が人工的マージに成功しており, マージ結果のソースコードが構文的な競合を含まないことが確認できれば, 人工的マージの適用時点より前方の編集の確認は不要である. これは, その時点より前方の編集は人工的マージのソースコードにすでに取り込まれており, 実際に開発者がマージを実行した時点で発生する構文的競合に無関係であることに基づいている. よって, 開発者は人工マージの適用時点より後方の編集操作だけを再演すればよい.

同様に考えると, もし人工的なマージにより生成されたソースコードが意味的な競合を含まないことを開発者が確認できれば, 人工的マージの適用時点より前方の編集操作は, 実際に開発者がマージを実行した時点で発生する意味的な競合に無関係である. よって, 意味的な競合を解決するために, ブランチに含まれるすべての編集操作を再演しようとする際にも, 開発者は人工マージの適用時点より後方の編集操作だけを再演すればよい.

このような前提をふまえて表 2 を見ると, 人工的マージと編集操作の並べ替えを適用することにより, 再演対象の編集操作の数が減少していることが分かる. たとえば, Main.java の「Total」の列を見ると, 構文的な競合を解決する際に注目する main() (cc₂) に関して, 再演対象の編集操作は 60 (385 – 325) 個減少しており, およそ 15% の減少率となっている. また, 意味的な競合を解決する際に調査する (と考えられる) すべてのクラスメンバ (All₂) に関して, 再演対象の編集操作の減少数は 170 (495 – 325) 個であり, およそ 34% の減少率となっている. これらの結果より, MergeHelper によって提示される編集操作をより重点的に被験者が再演することで, 効率的な競合解決が期

待できる。

ただし、編集操作数の減少が競合解決における被験者の手間の削減を直接示しているわけではない点も述べておく。なぜなら、編集操作の一部を再演対象から除外するためには、人工的マージにより生成されたソースコードが構文のおよび意味的な競合を含まないことを確認する作業が開発者に求められるからである。ここで、ソースコードが構文的な競合を含まないことは、構文解析（検査）器を用いることで、比較的容易に確認することができる。これに対して、ソースコードの意味的な正しさを厳密に検査する作業は困難であるため、ソースコードが意味的な競合を含まないと見なせる条件が必要になる。たとえば、今回の実験では、5.2 節で述べたように、「マージ後のソースコードがコンパイル可能かつ実行時例外で停止せずに動作する」ことを、競合が解決した条件とすることを被験者に伝えている。そのため、被験者は人工的なマージにより生成されたソースコードがこの条件を満たすことを随時検査することで、意味的な競合を含まないことを確認している。他にも、Guimarães らの手法 [5] のように、あらかじめ用意したテストケースを利用して、意味的な競合が含まれないことを確認することが考えられる。

Board.java については、競合する `restart()` (`cc1`) に関する編集操作がそれぞれのブランチにおいてもともと少なく（5 個と 8 個）、構文的な競合を解決する際に、編集操作数の減少の効果は期待できない。これに対して、すべての編集操作を再演する可能性のある意味的な競合解決においては（ All_2 では）、編集操作の減少数は 147 (160–13) 個（減少率はおおよそ 90%）となり、編集操作の並べ替えが有効に作用しているように見える。ただし、被験者のブランチ r に存在する（開発者 R が作成した） S_{3016} の Board.java と、Git のマージにより生成された S_{git} の Board.java における行差分は 1 行（開発者 N がブランチ n において挿入した代入文）だけであり、さらに Git のマージに成功していることから、競合の解決の際に開発者が編集操作の再演に時間を割く可能性は低い。実際、今回の実験では、被験者 K は 30 秒程度、被験者 T は 90 秒程度しか、Board.java の編集操作の再演には時間をかけていない。よって、Board.java における競合の解決に関して、MergeHelper が提示する再演対象の編集操作数の減少の効果は少なかったといえる。

5.5 インタビュー結果

実験手順 (4) のインタビューにおいて、各被験者に質問した事項を次に示す。

- 相手側の編集内容に関する理解について
- 競合に関する編集操作の抽出について
- 人工的マージや編集操作列の並べ替えについて

質問および回答は被験者と著者の 1 人が対面して、口頭

で行った。以下に、それぞれの項目に対する被験者 K および T からの回答をまとめる*6。

(1) 相手側の編集内容の理解について

インタビューでは、MergeHelper を用いた編集操作の再演により、相手側の開発者 N が実施した編集内容が理解できたかどうかを質問した。

被験者 K および T の両者とも、フィールド変数の追加に加えて、EXTRACTMETHOD リファクタリングが適用されていることに気付いていた。また、両者とも「編集内容を理解するうえで、編集履歴が大変役立った」と回答している。特に、「編集操作を逐次再演し、その際のコードの変化を追跡することで、EXTRACTMETHOD により新規にメソッドが抽出されていることを容易に把握できた」という回答が得られた。また、両者とも、「EXTRACTMETHOD の適用の理由はメソッドの肥大化である」と答えている。

このように編集履歴が編集の理解に約立つという意見を述べている一方、被験者 K は編集操作を逐次再演することが面倒であったことも指摘している。「今回の実験のように比較的小規模な編集履歴では大きな問題にはならないかもしれないが、競合に関係する編集履歴の規模が大きくなった場合には、編集操作の再演よりもソースコードの差分を利用することが現実的である」という意見を述べている。

(2) 競合に関する編集操作の抽出について

競合に関する可能性がある編集操作だけを MergeHelper が抽出することで、編集履歴全体を見ることに比べて、競合に関する理解作業が容易になったかどうかを質問した。

被験者 K は、「一部の編集操作が抽出されることで、最低限でも抽出された編集操作だけは把握しようという気持ちになった」と回答している。また、「抽出された編集操作を再演することで、競合の理解作業にスムーズにとりかかれた」と述べている。そのうえで、「もし競合に関係する編集操作だけを正確に抽出できるのであれば有用である」という感想を残している。

被験者 T は、「競合に関係すると見なした編集操作だけを追跡することで、大雑把に編集内容を把握できたことが良かった」、「競合の理解作業を始めるとっかかりとして有用である」と回答している。ただし、「最終的には編集履歴全体を確認した」とも述べている。

(3) 人工的マージや編集操作列の並べ替えについて

MergeHelper の提供する人工的マージや編集操作列の並べ替えによって、競合に関する理解作業が容易になったかどうか、さらに競合に関する理解の促進に役に立ったかどうかを質問した。

被験者 K および T の両者とも、「人工的マージにより競合に関係する可能性のある編集操作が削減されたり、並べ

*6 被験者は本論文の文章そのまの発言をしたわけではない。インタビューの際に書き留めたメモに基づき、被験者の発言の趣旨が変わらないように注意して発言を整理した。

替えによりなるべく多くの編集操作が反映されたマージ結果が実際に見られたりしたことで、競合に関する理解の手助けになった」と述べている。特に、被験者 K は、「競合に関係があると見なされた編集操作の数が少なくなることで、理解作業を進めるうえで気持ち的に楽になり、作業自体が快適になった」、「編集履歴の規模が大きい場合には、特に有用である」と述べている。

一方、両者とも、「実際の編集内容に関してある程度理解してからでないと、人工的に生成されたソースコードや編集操作の並べ替えはかえって作業を混乱させ、競合に関して間違った理解を誘発する可能性がある」と指摘している。特に被験者 T は、「編集内容を大まかに把握した後で、人工的マージにより生成されたソースコードを見たとき、最初の理解と目の前にあるソースコードとのギャップに戸惑った。具体的には、人工的に生成されたソースコードに突然現れたフィールド変数について、いつ誰が追加したのかという点で混乱した。そのため、人工的マージをいったん無効にして編集履歴の確認を行った」と答えている。

5.6 インタビューから得られた知見と考察

今回の実験における2名の被験者のインタビュー結果を見る限り、競合する2つのブランチの編集履歴を利用することで、それぞれの開発途中の編集の理解が容易になったと被験者は感じていることが分かる。特に、版管理システムに記録されるソースコードからだけでは推測しにくい編集の意図（今回の実験では、適用されたリファクタリングとそれが適用された理由）について、被験者が理解していた点は大きい。これらより、競合解決の最初の段階において必須である、過去の編集内容の把握に対して MergeHelper が役に立ったことが確認できた。

今回の実験を通して、MergeHelper の人工的マージおよび編集操作の並べ替えにより、再演対象として提示する編集操作の数が減少することが確認できた。また、定性的評価という観点からインタビュー項目 (3) の結果を見ても、編集操作の数の減少に関する効果について肯定的な意見が得られている。その一方で、人工的マージや編集操作列の並べ替えが競合に関する理解の妨げになる可能性があることが指摘されていることは無視できない。これに関しては、MergeHelper の使用性を改善し、人工的マージや編集操作列の並べ替えによる影響を分かりやすく提示することが重要である。

また、インタビューにおいて被験者から、「再演対象の粒度が細かすぎるのではないか」、「タイピングミスなどは再演しなくてもよい」という指摘があった。実際に競合を解決する際、開発者は文や式を単位としてソースコードを書き換えていることが多い。また、連続的に実行された編集操作をまとめて1つの編集と見なすことも多い。そこで、特定の文や式ごと編集操作を集約したり、編集時刻の近さ

に基づき編集操作を集約したりすることが考えられる [19]。他にも、理解の程度に応じて、再演したい編集操作の範囲や粒度を柔軟に指定したいという需要や、人工的マージを優先する範囲を自由に設定したいという需要が考えられる。競合解決を実施する開発者と MergeHelper との対話をより洗練し、競合の理解とその解決を段階的に支援する仕組みを確立することが重要である。

5.7 妥当性の脅威

今回の実験で扱った競合は単一、かつ擬似的に発生させたものである。また、それぞれのブランチにおいて競合の前に実施されたコミットは1回ずつであり、競合が発生するソースコードもあらかじめ被験者に与えられている。このように限定された場面ではなく、より一般的な場面において開発者が利用している競合の解決手法との比較を行うためには、数多くの評価実験の実施が必須である。残念ながら、MergeHelper が扱うことのできる細粒度のコード編集履歴を持ち、さらに競合するソースコードを含むブランチは現時点で存在しない。このため、通常の開発において、競合を発生させるコード編集履歴をより数多く蓄積するしかないのが現状である。

さらに、今回の実験において、被験者は2名である。事例の数が少ないことより、実験結果が被験者の個人的な特性や経験に大きく依存した可能性は排除できない。また、今回の実験において、被験者は自分で行った編集をマージ対象としたわけではなく、あくまでも開発者 R の立場でマージを実施している。このため、発生した競合を理解する以前に、開発者 R が行った編集内容に対する理解が十分でなかった可能性がある。一方、開発者は自分が過去に行った編集内容を必ずしも正確に把握しているわけではないため、今回の実験の設定には妥当性があるともいえる。いずれにしても、MergeHelper の有用性に関する評価に一般性を持たせるためには、競合に係る細粒度の編集履歴のさらなる収集と被験者実験が必須である。

MergeHelper における現在の実装では、ChangeTracker で記録した編集操作をそのまま再演の単位としている。しかしながら、ChangeTracker はできるだけ正確かつ細粒度で編集操作を記録することを目的としており、その粒度が競合の理解に適している根拠はない。このため、実装における妥協策が、今回の実験結果における効果に大きな影響を与えている可能性がある。

6. 関連研究

ここでは、マージ手法の改良による競合の解決支援、変更情報の共有による競合の回避支援、編集操作の再演による競合の解決支援という観点で、既存の手法やツールを取り上げる。

6.1 マージ手法の改良による競合の解決支援

Git のような版管理システムにおける 3 者間マージで発生する競合の回避を目的として、マージ手法を改良する研究が従来から行われている。Guimarães らは、ソースコードに含まれるクラスやメソッド・フィールド変数を木構造で表現し、それぞれの型やアクセスレベルなどを木構造の各節点の属性とすることで、属性単位でマージを試みる手法を提案している [5]。この手法では、それぞれのブランチで同じ属性が変更されていないか、マージ後のソースコードがコンパイル可能でありテストコードが通るか、クラス間の継承関係は正常かなどを判定することで、競合を検出する。Apel らは、マージを試みる開発者がプログラム要素ごとに行単位のマージ (unstructured merge) を行うのか、あるいはプログラムの構造を考慮したマージ (structured merge) を行うのかを自由に選択できる semi-structured merge を提案している [6], [20]。

これらのマージ手法の改良はどれも、マージ対象のソースコードにおいて、競合するプログラム要素の範囲をより狭く特定することを目的としている。これに対して、Lippe らは、ソースコードの時間的変化に着目し、コードの変換操作 (編集操作) に基づくマージ手法を提案している [7]。また、Shapiro らは、テキストエディタの編集ログに記録された各アクション (本論文における編集操作に相当) に事前事後条件を付与し、マージ後のテキストに制約を与えることで、利用者の意図を満たすマージ結果が得られるように編集の順番を自動的に決定する手法を提案している [21]。さらに、変換操作に着目するという観点では、Dig らが、リファクタリングの適用に限定して競合を解決する手法を提案している [22]。この手法では、競合に関係するリファクタリングを反転して適用することで、リファクタリング適用前のソースコードを復元する。そのソースコードに対して 3 者間マージを実施した後、もとのリファクタリングを再適用することで、競合の解決を達成している。

マージ手法の改良により競合が解決される可能性は高まるが、競合を完全に排除することはできず、さらには開発者がマージ結果をつねに受け入れるとは限らない。このため、手動による競合解決の支援は必須である。残念ながら、従来の手法はどれも競合解決を主な目的としており、競合の理解や競合の発生原因の調査という観点での支援が不十分である。MergeHelper では競合の解決支援において、編集操作の再演を取り入れている点が大きな特徴である。

6.2 変更情報の共有による競合の回避支援

競合を直接解決するのではなく、編集を開発者間で共有したり、競合をできるだけ早期に検出したりすることで、競合の発生の回避や競合解決のための手戻りの削減を目指す研究も行われている。たとえば、他の開発者の編集をリアルタイムに気付かせる (コードの編集を共有する) 仕組

みを導入したツールとして、FastDash [23], Palantir [24], CollabVS [25], Syde [26] がある。Syde は、編集集中のソースコードの抽象構文木において、節点単位での挿入/削除、節点に対するプロパティの挿入/削除/変更をリアルタイムに検知し、それらの変更と発生する可能性のある競合を開発者間で共有する仕組みを提供している [26], [27]。

絶え間ない (連続的な) マージを目指したツールとしては、Crystal [28] や WeCode [5] がある。Crystal は、各開発者のローカルリポジトリとリモートリポジトリのバージョン情報に基づき、現在編集集中のソースコードをマージした場合の結果を逐次計算し、それを開発者に通知する [28], [29]。WeCode は、開発者が編集集中のソースファイルを保存した時点およびコンパイルに成功した時点において、バックグラウンドで競合の検出を行い、その結果を開発者に報告する [5]。さらに、コード編集の共有と連続的なマージを統合したツールに CloudStudio [30], [31] がある。このツールは、各開発者の編集ごとにマージが実行される協調作業環境を提供する。

これらのツールを利用することで、競合の発生をほぼリアルタイムに知ることができる。その一方で、競合が通知されたからといって、開発者はただちに競合の解決にとりかかるとは限らない。よって、解決を先延ばしにした競合に対する理解や、その競合を解決するための支援が不要になるわけではない。MergeHelper は競合を検出した後でその解決を支援するツールであり、これらのツールと利用状況が異なる。ただし、排他的な利用を想定しているわけではなく、組み合わせて利用するのが現実的である。

6.3 編集操作の再演による競合の解決支援

細粒度の編集操作を記録し、それらを再演することで、過去のソースコード変更の理解を支援するツールはすでに存在する [32]。著者らも OperationRecorder [18] と OperationReplayer [33] を提案しており、これらを組み合わせることで編集操作の記録と再演は実現できる。さらに、OperationSliceReplayer [11] を利用することで、コード編集履歴全体から特定のクラスメンバの変更に係る編集操作だけを効率的に再演することも可能である。ここで、各編集操作の実行前後のソースコードの変化を視覚化するという観点で、既存の編集操作再演ツールの再演機能はほぼ同じである。そこで、MergeHelper と同じ再演機能を備えた OperationReplayer およびその拡張である OperationSliceReplayer との比較を通して、MergeHelper の優位点を述べる。

編集操作の再演により開発者の過去の変更にに対する理解を促進するという観点で、3 つの再演ツールに違いはない。いいかえれば、3 つの再演ツールにおいて、再演対象の編集操作が同一であれば、競合の解決に費やす開発者の手間は変わらないはずである。これに対して、MergeHelper の

人工的マージは、ブランチに含まれる一部の編集を除外することで、OperationReplayer および OperationSliceReplayer に比べて、開発者が再演対象として意識する編集操作の数を減少させることができる。

このことは、表 2 を見ることで確認できる。「Original」の行の「cc₁」および「cc₂」の列が OperationSliceReplayer を利用した場合、「All₁」および「All₂」の列が OperationReplayer を利用した場合における再演対象の編集操作の数と同じ値を指している。これに対して、「AMerge」と「AMerge+Swap」の行が MergeHelper を利用した場合の数を指す。また、今回の実験では、OperationReplayer や OperationSliceReplayer を単独で利用した場合との比較実験は行っていない。しかしながら、5.5 節のインタビュー項目 (3) における肯定的な意見を見る限り、MergeHelper が OperationReplayer や OperationSliceReplayer に比べて有用である場面が存在するといえる。

7. おわりに

マージにおける競合の解決は開発者にとって面倒な作業である。本論文では、細粒度のコード編集履歴を用いることで、版管理システム Git の利用時に発生する競合の解決を支援するツール MergeHelper を提案した。また、被験者実験を通して、MergeHelper の有用性と残された課題を示した。MergeHelper は Web サイト^{*7}より入手可能である。

今後の課題として、まずは MergeHelper の使用性の改善を予定している。たとえば、人工的マージにより大幅に見た目が変わるコード断片や並べ替えられた編集操作列に色付けするなど、もとのソースコードや編集操作と容易に区別できることが望まれる。さらに、MergeHelper の機能拡張として、連続するマージへの対応を考えている。現在の実装では、2つのブランチのソースコードがマージに成功した場合でも、編集履歴のマージは行っていない。さらに、Git の出力する図 2 のようなソースファイルにおいて開発者が競合の解決を行った場合にも、収集した編集操作をもとのブランチに反映させていない。よって、自動あるいは手動にかかわらず、マージにより生成されたソースコードの編集履歴は不完全なままである。このような状況で連続してマージを適用すると、MergeHelper は正しく動作しない。これを解決するためには、コード編集履歴を適時加工したり、適切に分散管理したりする仕組みの検討が必須である。

さらなる評価実験の実施も重要な課題である。しかしながら、競合するソースコードの編集履歴はきわめて少なく、その蓄積を待つだけでは不十分である。そこで、ChangeDistiller [8] や ChangeNodes [34] を利用することで、オープンソースプロジェクトの中で実際に競合が発生したソース

コードとものソースコードの差分から、擬似的にコード変更（編集操作）を抽出することを考えている。また、評価実験においては、競合の解決パターンによる分類が必要になる可能性がある。その際、メソッド間の競合がどのように解決されているのかを調査した Yuzuki らの研究 [35] が参考になると考えている。

謝辞 OperationRecorder により記録した実験データを提供していただいた立命館大学情報理工学部の大森隆行氏に感謝する。本研究の一部は、科研費 (15H02685) の助成を受けたものである。

参考文献

- [1] Perry, D.E., Siy, H.P. and Votta, L.G.: Parallel Changes in Large-scale Software Development: An Observational Case Study, *ACM TOSEM*, Vol.10, No.3, pp.308–337 (2001).
- [2] O'Sullivan, B.: Making Sense of Revision-Control Systems, *CACM*, Vol.52, No.9, pp.56–62 (2009).
- [3] Phillips, S., Sillito, J. and Walker, R.: Branching and Merging: An Investigation into Current Version Control Practices, *Proc. CHASE '11*, pp.9–15 (2011).
- [4] Mens, T.: A State-of-the-Art Survey on Software Merging, *IEEE Trans. Softw. Eng.*, Vol.28, No.5, pp.449–462 (2002).
- [5] Guimarães, M.L. and Silva, A.R.: Improving Early Detection of Software Merge Conflicts, *Proc. ICSE '12*, pp.342–352 (2012).
- [6] Apel, S., Liebig, J., Brandl, B., Lengauer, C. and Kästner, C.: Semistructured Merge: Rethinking Merge in Revision Control Systems, *Proc. ESEC/FSE '11*, pp.190–200 (2011).
- [7] Lippe, E. and van Oosterom, N.: Operation-based Merging, *SIGSOFT Softw. Eng. Notes*, Vol.17, No.5, pp.78–87 (1992).
- [8] Fluri, B., Wüersch, M., Pinzger, M. and Gall, H.: Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction, *IEEE Trans. Softw. Eng.*, Vol.33, No.11, pp.725–743 (2007).
- [9] Soetens, Q.D., Robbes, R. and Demeyer, S.: Changes As First-Class Citizens: A Research Perspective on Modern Software Tooling, *ACM Comput. Surv.*, Vol.50, No.2, pp.18:1–18:38 (2017).
- [10] Negara, S., Vakilian, M., Chen, N., Johnson, R.E. and Dig, D.: Is It Dangerous to Use Version Control Histories to Study Source Code Evolution?, *Proc. ECOOP '12*, pp.79–103 (2012).
- [11] Maruyama, K., Kitsu, E., Omori, T. and Hayashi, S.: Slicing and Replaying Code Change History, *Proc. ASE '12*, pp.246–249 (2012).
- [12] Maruyama, K., Omori, T. and Hayashi, S.: Slicing Fine-Grained Code Change History, *IEICE Trans. Inf. Syst.*, Vol.99, No.3, pp.671–687 (2016).
- [13] Servant, F. and Jones, J.A.: Fuzzy Fine-grained Code-history Analysis, *Proc. ICSE '17*, pp.746–757 (2017).
- [14] Hayashi, S., Omori, T., Zenmyo, T., Maruyama, K. and Saeki, M.: Refactoring Edit History of Source Code, *Proc. ICSM '12*, pp.617–620 (2012).
- [15] Hayashi, S., Hoshino, D., Matsuda, J., Saeki, M., Omori, T. and Maruyama, K.: Historef: A tool for Edit History Refactoring, *Proc. SANER '15*, pp.469–473 (2015).

^{*7} <http://www.fse.cs.ritsumei.ac.jp/mergehelper/>

- [16] Herzig, K. and Zeller, A.: The Impact of Tangled Code Changes, *Proc. MSR '13*, pp.121–130 (2013).
- [17] Hayashi, S. and Saeki, M.: Recording Finer-Grained Software Evolution with IDE: An Annotation-Based Approach, *Proc. IWPSE-EVOL '10*, pp.8–12 (2010).
- [18] Omori, T. and Maruyama, K.: A Change-Aware Development Environment by Recording Editing Operations of Source Code, *Proc. MSR '08*, pp.31–34 (2008).
- [19] Kitsu, E., Omori, T. and Maruyama, K.: Detecting Program Changes from Edit History of Source Code, *Proc. APSEC '13*, pp.299–306 (2013).
- [20] Apel, S., Leßenich, O. and Lengauer, C.: Structured Merge with Auto-Tuning: Balancing Precision and Performance, *Proc. ASE '12*, pp.120–129 (2012).
- [21] Shapiro, M., Rowstron, A. and Kermarrec, A.-M.: Application-independent reconciliation for nomadic applications, *Proc. SIGOPS European Workshop*, pp.1–6 (2000).
- [22] Dig, D., Manzoor, K., Johnson, R. and Nguyen, T.N.: Refactoring-Aware Configuration Management for Object-Oriented Programs, *Proc. ICSE '07*, pp.427–436 (2007).
- [23] Biehl, J.T., Czerwinski, M., Smith, G. and Robertson, G.G.: FASTDash: A Visual Dashboard for Fostering Awareness in Software Teams, *Proc. CHI '07*, pp.1313–1322.
- [24] Sarma, A., Bortis, G. and van der Hoek, A.: Towards Supporting Awareness of Indirect Conflicts Across Software Configuration Management Workspaces, *Proc. ASE '07*, pp.94–103 (2007).
- [25] Hegde, R. and Dewan, P.: Connecting Programming Environments to Support Ad-Hoc Collaboration, *Proc. ASE '08*, pp.178–187 (2008).
- [26] Hattori, L. and Lanza, M.: Syde: A Tool for Collaborative Software Development, *Proc. ICSE '10*, pp.235–238 (2010).
- [27] Lanza, M., Hattori, L. and Guzzi, A.: Supporting Collaboration Awareness with Real-Time Visualization of Development Activity, *Proc. CSMR '10*, pp.202–211 (2010).
- [28] Brun, Y., Holmes, R., Ernst, M.D. and Notkin, D.: Proactive Detection of Collaboration Conflicts, *Proc. ESEC/FSE '11*, pp.168–178 (2011).
- [29] Brun, Y., Holmes, R., Ernst, M.D. and Notkin, D.: Crystal: Precise and Unobtrusive Conflict Warnings, *Proc. ESEC/FSE '11*, pp.444–447 (2011).
- [30] Estler, H.-C., Nordio, M., Furia, C.A. and Meyer, B.: Unifying Configuration Management with Merge Conflict Detection and Awareness Systems, *Proc. ASWEC '13*, pp.201–210 (2013).
- [31] Estler, H.-C., Nordio, M., Furia, C.A. and Meyer, B.: Awareness and Merge Conflicts in Distributed Software Development, *Proc. ICGSE '14*, pp.26–35 (2014).
- [32] 大森隆行, 林 晋平, 丸山勝久: 統合開発環境における細粒度な操作履歴の収集および応用に関する調査, コンピュータソフトウェア, Vol.32, No.1, pp.60–80 (2015).
- [33] Omori, T. and Maruyama, K.: An Editing-operation Replayer with Highlights Supporting Investigation of Program Modifications, *Proc. IWPSE-EVOL '11*, pp.101–105 (2011).
- [34] Stevens, R. and Roover, C.D.: Extracting Executable Transformations from Distilled Code Changes, *Proc. SANER '17*, pp.171–181 (2017).
- [35] Yuzuki, R., Hata, H. and Matsumoto, K.: How We Resolve Conflict: An Empirical Study of Method-Level

Conflict Resolution, *Proc. SWAN '15*, pp.21–24 (2015).



西村 雄一

2015 年立命館大学情報理工学部情報システム学科卒業。2017 年同大学大学院理工学研究科博士課程前期課程修了。同年日立金属株式会社入社。学生時代は、ソフトウェア進化の研究に従事。



紙名 哲生 (正会員)

1999 年国際基督教大学教養学部卒業。2002 年東京大学大学院総合文化研究科修士課程修了。2005 年東京大学大学院総合文化研究科博士後期課程修了。2005 年 4 月より筑波大学先端学際領域研究センター研究員。2006 年 4 月より東京大学大学院人文社会系研究科助手。2007 年 4 月より助教。2010 年 4 月より同大学院教育学研究科特任助教。2014 年 4 月より立命館大学情報理工学部講師。プログラミング言語やソフトウェア工学、特にモジュール化機構に関する研究に従事。博士 (学術)。日本ソフトウェア科学会, ACM 各会員。



丸山 勝久 (正会員)

1991 年早稲田大学理工学部電気工学科卒業。1993 年同大学大学院理工学研究科修士課程修了。同年日本電信電話株式会社入社。2000 年 4 月より立命館大学理工学部助教授。2007 年 4 月より同大学情報理工学部教授。2003 年 9 月～2004 年 9 月, University of California, Irvine 客員研究員。ソフトウェア保守と進化, ソフトウェア再利用, ソフトウェア開発環境, プログラム理解支援の研究に従事。博士 (情報科学)。電子情報通信学会, 日本ソフトウェア科学会, IEEE-CS, ACM 各会員。