プログラム編集履歴を用いた版管理システムでの競合解決支援

西村 雄一　丸山 勝久

ソフトウェア開発において，版管理システムのマージ機能を利用することで並行開発が容易になる．しかしながら，2 つのブランチを統合する際の競合は避けられない．手動による競合の解決では，それぞれのブランチにおいて実施されたすべての編集を把握する必要があり，その手間は大きい．本論文では，それぞれのブランチに存在するプログラムの編集履歴を用いることで競合を検出し，競合に関係のある編集操作だけを抽出する手法を提案する．

1　はじめに

効率的なソフトウェア開発や保守を実現するため，Subversion[†1] や Git[†2] といった分散型の版管理システムを用いることが一般的になりつつある．これらの版管理システムはソースコードのブランチを作成したり，それらをマージしたりする機能を備えている．これらの機能を用いることで，最新版やリリース版のソースコードに影響を及ぼすことなくそれぞれの開発者が独自にソースコードを改変することができる．さらには，独自に改変したソースコードの差分をもとの版に統合（反映）することも可能である．このようにして，複数の開発者による分散開発を支援している．

しかしながら，このような分散開発が常にうまくいくとは限らない．ある開発者が改変したコード断片が，別の開発者が改変したコード断片と重なっていた場合，それらのコード断片は競合状態となる．このようなコード断片は互いに干渉するため，版管理システムのマージ機能により単純に統合することはできない．このような競合を解決するためには，後から統合を試みた開発者が競合しているコード断片を修正する必要がある．残念ながら，現在の版管理システムを利用している限りにおいて，開発者が競合に関係するコード断片と関係しないコード断片を特定する必要がある．さらに，それぞれのソースコード改変において実施された編集を理解した上で，ソースコードを修正する必要がある．一般的に，このような修正作業は面倒であり，時間がかかる．また，修正対象のソースコードに対する理解が不十分なまま修正を行うと，予期せぬ動作の発生やバグの混入につながる．

本論文では，開発者がどのようにプログラムを変更してきたを保存するプログラム編集履歴を用いて，ソースコードの統合における競合の解決を支援する手法を提案する．本手法では，統合対象のブランチにおいて，それぞれ編集されたソースコードの編集操作が存在することを前提とする．また，編集が行われたソースコードに現れるメソッドやフィールドと編集操作の関係を表現した編集操作グラフ (edit operation graph) [4] を利用する．さらに，編集操作グラフに基づき，特定のプログラム要素の構築に関係のある編集操作だけを抽出する編集履歴スライス (operation history slice) [4] を利用することで競合を検出する．最終的に，競合に関係のあるメソッドおよびフィールドに関係する編集操作だけを抽出し，それらの編集を再演することで競合の解決を支援する．競合に関係

Supporting Conflict Resolution in Version Control Systems by Using Edit Operation History

Yuichi Nishimura, 立命館大学大学院情報理工学研究科, Graduate School of Informamtion Science and Engineering, Ritsumeikan University.

Katsuhisa Maruyama, 立命館大学情報理工学部, Department of Computer Science, Ritsumeikan University.

†1　http://subversion.apache.org/

†2　http://git-scm.com/

ある一部の編集操作だけを再演するだけでよいので，開発者の手間の削減が期待できる．

2 プログラム編集履歴を用いた競合解決

ここでは，まず 2 つのブランチを統合する際に発生する競合の例を示す．次に，競合の解決を支援するために，競合に関係のあるメソッドおよびフィールドと，競合に関係する編集操作を特定する手法を提案する．

2.1 競合の例

いま，Git を用いた共有リポジトリをブランチ名 master で管理する開発者 M がいる．このリポジトリには，図 1 に示す Book クラスを含むファイル Book.java のソースコード（M_0 と呼ぶ）が格納されている．

このような状況において，別の開発者 B が名前 branch を持つ新たなブランチを共有リポジトリに作成し，M_0 をチェックアウトすることで得られたソースコード B_0 の Book に対して，次に示す手順で編集を行った．

(b1) メソッド getPrice() のコピーを作成

(b2) 2 つの getPrice() のうちのひとつの名前を getPriceValue() に変更

(b3) メソッド getPrice() の返り値の型を int から String に変更

(b4) price の値の末尾に通貨単位 “-yen” を付与したものを返すように getPrice() の本体を書き換え

修正後のソースコード（B_4 と呼ぶ）を図 2 に示す．

開発者 B の作業と同時に開発者 M も開発を継続しており，ソースコード M_0 の Book に対して，次に示す手順で編集を行った．

(m1) Book のフィールド変数 price の型を int から String に変更

(m2) price の値として，価格の末尾にその通貨単位 “-yen” を付与したものを格納するようにコンストラクタ Book(...) の本体を書き換え

(m3) メソッド getPrice() の返り値の型を int から String に変更

(m4) getAuthors() の戻り値の型を String から String[] に変更

(m5) コンマで区切られた String 型の著者リストを分割し，それぞれの著者を String の配列で返すように getAuthors() の本体を書き換え

修正後のソースコード（M_5 と呼ぶ）を図 3 に示す．

このような並行開発において，まず開発者 M が

```
public class Book {
  private String title;
  private String authors;
  private int price;

  public Book(String t, String a, int p) {
    title = t;
    authors = a;
    price = p;
  }

  public String getTitle() {
    return title;
  }

  public String getAuthors() {
    return authors;
  }

  public int getPrice() {
    return price;
  }
}
```

図 1 ブランチ master の Book クラス (M_0)

```
public class Book {
  private String title;
  private String authors;
  private int price;

  public Book(String t, String a, int p) {
    title = t;
    authors = a;
    price = p;
  }

  public String getTitle() {
    return title;
  }

  public String getAuthors() {
    return authors;
  }

  public String getPrice() {
    return String.valueOf(price) + "-yen";
  }

  public int getPriceValue() {
    return price;
  }
}
```

図 2 ブランチ branch の Book クラス (B_4)

```
public class Book {
  private String title;
  private String authors;
  private String price;

  public Book(String t, String a, int p) {
    title = t;
    authors = a;
    price = String.valueOf(p) + "-yen";
  }

  public String getTitle() {
    return title;
  }

  public String[] getAuthors() {
    return authors.split(",");
  }

  public String getPrice() {
    return price;
  }
}
```

図 3　ブランチ master の Book クラス (M_5)

M_5 を master にコミットした．その後，開発者 B が B_4 を branch にコミットし，branch に行った編集を master に統合しようと試みた場合を考える．この場合，競合が発生し，統合に失敗する．

2.2　手動による競合の解決

手動での統合を実施するために，ブランチ master に適用された編集を B_4 にマージすると，次に示すような競合が報告される（ブランチ branch の B_4 が以下のような記述を含む）．

```
  public String getPrice() {
<<<<<<< HEAD
    return String.valueOf(price) + "-yen";
=======
    return price;
>>>>>>> refs/remotes/origin/master
  }
```

上記の報告により，`getPrice()` に対する編集が被っていることが分かる．しかしながら，上記の報告だけを見て，単純に競合箇所を修正するのは危険である．たとえば，`getPrice()` の本体を，以下に示すコード断片

```
return String.valueOf(price) + "-yen";
```

のように単純に修正してしまうと，コミットされたソースコードがコンパイルエラーを発生させたり，予期せぬ動作を行う可能性がある．

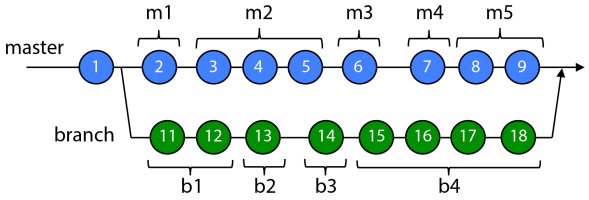

図 4　ブランチ master と branch の編集操作

この例では，`getPriceValue()` の戻り値の型が一致しない．また，`getPrice()` により返される文字列において，通貨単位が重複する．

もし，`getPrice()` の本体を上記のように修正したいのであれば，同時に変更 (m1)(m2) を取り消すようにソースコードを書き換えるのが望ましい．つまり，2 つのブランチを統合する際には，競合しているコード断片だけを修正すればよいわけではなく，それぞれのブランチで適用されたすべての編集を把握した上で，競合を解決する必要がある．

2.3　支援手法

本論文では，開発者がどのようにプログラムを変更してきたのかを表現したプログラム編集履歴を用いて，ソースコードの統合における競合の解決を支援する手法を提案する．本手法では，統合対象の 2 つのブランチにおいて，それぞれ編集されたソースコードの編集操作が存在することを前提とする．編集操作の取得には，筆者らが開発した Eclipse[†3] プラグインの ChangeTracker[†4] を活用する．2.1 節の例における編集操作を図 4 に示す．

ここで，ChangeTracker は，ファイルごとの編集操作を時間順に再演（再生）する機能を備えている．よって，それぞれのブランチに存在するソースコードを統合する場合は，それらの編集操作をそれぞれ再演することで，編集の把握が容易になる．ただし，この

†3　http://www.eclipse.org/

†4　Fluorite [6] と同様に Eclipse の Java エディタ上で実施されたテキスト編集イベントを捕獲し，OperationRecorder [5] と同様の XML 形式で編集履歴を出力する．

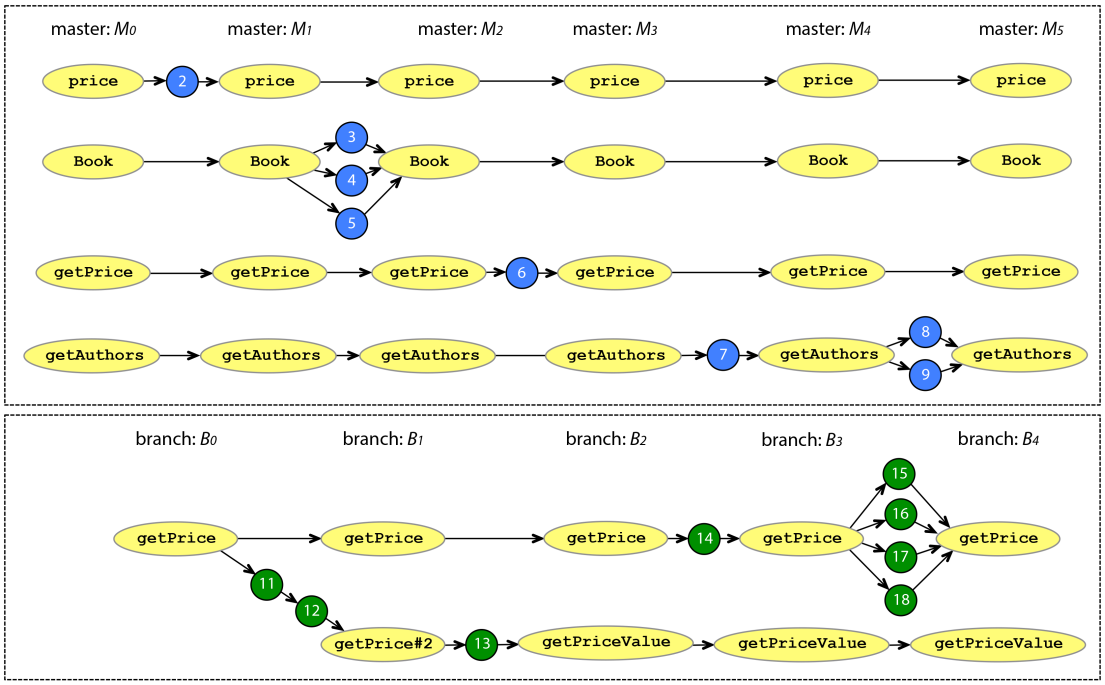

図 5　ブランチ master と branch のソースコードに対する編集操作グラフ（一部のみ）

ツールは単一のファイルに対する編集操作を単純に再演できるだけであり，競合の検出や競合の解決を十分に支援しているとはいえない．

そこで，ChangeTracker を拡張し，競合解決に関係のある編集操作だけを抽出し，それらを再演することで，競合解決の手間の削減を目指す．そのために，それぞれのブランチに対応するプログラム編集履歴から編集操作グラフを構築する．このグラフは，各編集操作と各プログラム要素（メソッドおよびフィールド）を節点として持つ．また，それぞれの編集操作は，それが編集しているプログラム要素に矢印で接続されている．さらに，本手法では，編集操作グラフに基づき，特定のプログラム要素の編集（記述）に関係のある編集操作だけを抽出する編集履歴スライスを利用する．

以下に，競合に関係する編集操作を求める手順を示す．

(Step 1)　プログラム編集履歴全体から，ブランチ作成時点から統合時点までの編集操作を抽出する．編集操作は時間順に並んでいるため，統合対象のブランチ branch が作成された時刻と統合を試みた時刻から，その間に適用されたすべての編集操作を集めることができる．図 4 の例では，編集操作 2 ～ 9 と編集操作 11 ～ 18 が抽出される．

(Step 2)　ブランチ b に含まれるファイル f ごとに，編集操作グラフ $G(b, f)$ を作成する．図 4 の編集操作に対応する編集操作グラフを図 5 に示す．

(Step 3)　ブランチ b のファイル f に関して，統合するソースコード s を解析し，そのソースコード内部で定義されているプログラム要素を抽出する．抽出したプログラム要素をそれぞれ $P(b, s)$ と表す．2.1 節の例では，次のようになる．

$P(\mathsf{master}, M_5)$
$= \{$ `title#Book`, `author#Book`,
`price#Book`, `Book#Book`,
`getTitle#Book`, `getAuthors#Book`,
`getPrice#Book` $\}$

$P(\mathsf{branch}, B_4)$
$= \{$ `title#Book`, `author#Book`,
`price#Book`, `Book#Book`,

getTitle#Book, getAuthors#Book,
getPrice#Book, getPriceValue#Book }

ただし，E#C はクラス C のプログラム要素 E を指す．

(Step 4) プログラム要素 $p \in P(b, s)$ を一つずつ選択し，$G(b, f)$ に基づき編集履歴スライス $S(b, p)$ を作成する．2.1 節の例では，ブランチ master と branch から，それぞれ 7 個および 8 個のスライスを作成する．

$S(\mathsf{master}, \texttt{title\#Book}) = \emptyset$
$S(\mathsf{master}, \texttt{authors\#Book}) = \emptyset$
$S(\mathsf{master}, \texttt{price\#Book}) = \{\ 2\ \}$
$S(\mathsf{master}, \texttt{Book\#Book}) = \{\ 3, 4, 5\ \}$
$S(\mathsf{master}, \texttt{getTitle\#Book}) = \emptyset$
$S(\mathsf{master}, \texttt{getAuthors\#Book}) = \{\ 7, 8, 9\ \}$
$S(\mathsf{master}, \texttt{getPrice\#Book}) = \{\ 6\ \}$
$S(\mathsf{branch}, \texttt{title\#Book}) = \emptyset$
$S(\mathsf{branch}, \texttt{authors\#Book}) = \emptyset$
$S(\mathsf{branch}, \texttt{price\#Book}) = \emptyset$
$S(\mathsf{branch}, \texttt{Book\#Book}) = \emptyset$
$S(\mathsf{branch}, \texttt{getTitle\#Book}) = \emptyset$
$S(\mathsf{branch}, \texttt{getAuthors\#Book}) = \emptyset$
$S(\mathsf{branch}, \texttt{getPrice\#Book}) = \{14, 15, 16, 17, 18\}$
$S(\mathsf{branch}, \texttt{getPriceValue\#Book}) = \{11, 12, 13\}$

(Step 5) 統合する 2 つのブランチを b_1 および b_2 とする．また，それぞれのソースコードを s_1 および s_2 とする．プログラム要素 $p1 \in P(b_1, s_1)$ および $p_2 \in P(b_2, s_2)$ に対して，ラベル（本手法ではメソッドの名前あるいはフィールドの名前）が一致する 2 つのスライス $S(b_1, p_1)$ と $S(b_2, p_2)$ を取り出す．本手法では，これら 2 つのスライスがともに空集合でないとき，b_1 の p_1 と b_2 の p_2 が競合すると見なす．2.1 節の例では，master の getPrice#Book) と branch の getPrice#Book が競合と判定される．

(Step 6) ブランチ b のプログラム要素 p が競合している場合，競合に関係のある編集操作の列は編集履歴スライス $S(b, p)$ に含まれる編集操作を時間順に並べたものとなる．本手法では，この編集操作の列を再演の対象とする．2.1 節の例では，master の M_5（Book.java）に対しては編集操作の列 [6]，branch の B_4（Book.java）に対しては編集操作の列 [14, 15, 16, 17, 18] が再演対象として特定される．

3 関連研究

競合が発生したことを知らせるものとして Syde[3] や WeCode[2] がある．これらは協調開発におけるアウェアネス[1] の改善を目指している．

Syde はクライアントサーバーアプリケーションであり，クライアントである Eclipse プラグインを通じてプログラムの変更履歴をサーバーに送信する．送信された変更履歴は，Abstract Syntax Tree (AST) のノード単位 (構文要素) での変更情報が含まれており，サーバー側はそれらの保存および変更された部分の情報・衝突の有無をクライアントにブロードキャストする．

WeCode は，版管理システムの一つである Subversion をサポートしており，複数の開発者・開発チームでの開発を想定している．開発者がプログラミングを行っている最中に，バックグラウンドでそのプログラムをサーバーへ送信しサーバー側で最新のコミットと比較する．この時，開発者に代わって競合となりうる部分を発見することで，開発者がコミットやマージを行う前に競合の有無を伝えることを可能にしている．

本手法の目的は，Syde や WeCode のように競合が発生したことを素早く知らせることではなく，競合を手動で解決する際に必須である編集に関する情報を提供することである．

4 おわりに

本論文では，版管理システムのひとつである Git において発生する競合の解決を，細粒度のプログラム編集履歴を用いて支援する手法を述べた．現在，2.3 節に示した手順を ChangeTracker に組み込んでいる途中である．

今後の課題として，実装を完成させ，評価実験に取り組む予定である．同時に，手法の改良も検討する．

本手法では，ファイル，クラス，メソッド，フィールドの名前変更を追跡しておらず，名前変更を含んだソースコードを統合する場合には，競合に関係ある編集操作が正しく抽出されない可能性が高い．従って，名前変更の追跡は必須である．また，本手法では，メソッドおよびフィールドを競合の単位と見なしている．実際には，メソッドの一部の編集が被った場合でも，競合を引き起こさない場合がある．このような状況に対応するために，編集操作に記録された編集範囲（ソースコード上のオフセットで管理されている）を活用することを考えている．

謝辞 本研究に関してご討論頂きました，立命館大学の紙名哲生氏に感謝いたします．本研究の一部は，科研費（24500050, 15H02685）の助成を受けたものである．

参 考 文 献

[1] Dourish, P. and Bellotti, V.: Awareness and Coordination in Shared Workspaces, *Proc. CSCW'92*, 1992, pp. 107–114.

[2] Guimarães, M. L. and Silva, A. R.: Improving Early Detection of Software Merge Conflicts, *Proc. ICSE'12*, 2012, pp. 342–352.

[3] Hattori, L. and Lanza, M.: Syde: A Tool for Collaborative Software Development, *Proc. ICSE'10*, 2010, pp. 235–238.

[4] Maruyama, K., Kitsu, E., Omori, T., and Hayashi, S.: Slicing and Replaying Code Change History, *Proc. ASE'12*, 2012, pp. 246–249.

[5] Omori, T. and Maruyama, K.: A Change-Aware Development Environment by Recording Editing Operations of Source Code, *Proc. MSR'08*, 2008, pp. 31–34.

[6] Yoon, Y. and Myers, B. A.: Capturing and Analyzing Low-level Events from the Code Editor, *Proc. PLATEAU'11*, 2011, pp. 25–30.