特別研究報告

題目

マージコンフリクト解消時に片側採用が選択された理由の調査

指導教員
井上 克郎 教授

報告者
林　大輝

令和 4 年 2 月 8 日

大阪大学 基礎工学部 情報科学科

令和 3 年度 特別研究報告

マージコンフリクト解消時に片側採用が選択された理由の調査

林　大輝

内容梗概

現代のソフトウェア開発においては，複数の開発者が並行して開発を進めることが多くある．開発者はまず本流のブランチから新たに自身が作業するための派生ブランチを作成し，派生ブランチ上で編集作業を行う．成果物が完成したら，派生ブランチの作業内容を本流のブランチに統合する．

ブランチを用いた作業を統合する際に問題となるのが，マージコンフリクトと呼ばれる現象である．派生ブランチの作業により編集された箇所が本流のブランチにおいても変更されていた場合，統合作業においてどちらの変更を残すべきか，機械的に判断することができない．これがマージコンフリクトである．ソフトウェア開発においてマージコンフリクトは頻繁に発生することが知られている．

具体的なマージコンフリクトの解消方法に関して，2 つの親コミットのどちらか片方の編集を採用し，もう片方の親の編集を削除することで，マージコンフリクトの解消が行われるケースが多く存在していることが知られている．これを片側採用といい，湯月らの研究によって，Java プロジェクト中のメソッド内部で発生したマージコンフリクトにおいては，全体の約 98% が片側採用で解消されていることが明らかになっている．

マージコンフリクトが片側採用により解消されるということは，並行して進められた開発成果のうち片側が切り捨てられるということでもある．これまでの研究では，マージコンフリクトの解消によるプロジェクトへの影響は多く議論されていないため，開発成果が本当に切り捨てられてよいものかなどの分析は行われていない．

本研究では，マージコンフリクトの解消時に片側採用が選択された理由を調査し，分類を行う．具体的には，Java で書かれた OSS プロジェクトの中から片側採用により解消されたマージコンフリクトを収集し，解消時に削除されたコードの内容や対となるブランチとの差異，コミットメッセージなどをもとに，形式的な違いで 8 種類に分類した．調査の結果，片側採用が選択されたマージコンフリクトでは，フォーマットやコメント，import 文に関するものが多いことが分かった．

主な用語

分散開発
マージコンフリクト
OSS 開発

目 次

1　まえがき

現代のソフトウェア開発においては，複数の開発者が並行して開発を進めることが多くある．同時並行的な開発作業を支援するツールとして，バージョン管理システム（Version Control System．以下 VCS）が頻繁に用いられる．多くの VCS では，ソフトウェアの過去の版を記録しておく機能のほか，複数人が並行して開発した成果を統合する作業を支援する仕組みが取り入れられている．例えば，開発者はまず本流のブランチから新たに自身が作業するための派生ブランチを作成し，派生ブランチ上で編集作業を行う．成果物が完成したら，派生ブランチの作業内容を本流のブランチに統合する．派生ブランチは同時に複数作成することができるため，複数の開発者が同時に作業を行ったとしても，統合作業を繰り返すことで効率的にソフトウェア開発を進めることができる．

ブランチを用いた作業を統合する際に問題となるのが，マージコンフリクトと呼ばれる現象である．派生ブランチの作業により編集された箇所が本流のブランチにおいても変更されていた場合，統合作業においてどちらの変更を残すべきか，機械的に判断することができない．これがマージコンフリクトである．VCS を用いたソフトウェア開発においてマージコンフリクトは頻繁に発生することが知られている．Burn らがオープンソースソフトウェア 9 個の開発履歴を調査した結果，すべてのプロジェクトにおいてマージコンフリクトの履歴があり，マージの数に対して平均約 19%の割合で発生していたことが明らかになった [5]．一方で，マージコンフリクトの解消には時間と手間がかかることが知られており，ソフトウェア開発プロジェクト全体においてコスト要因となりうる [8]．

また，具体的なマージコンフリクトの解消方法に関して，2 つの親コミットのどちらか片方の編集を採用し，もう片方の親の編集を削除することで，マージコンフリクトの解消が行われるケースが多く存在していることが知られている．これを片側採用といい，湯月らの研究によって，Java プロジェクト中のメソッド内部で発生したマージコンフリクトにおいては，全体の約 98% が片側採用で解消されていることが明らかになっている [18]．白木らはこの結果をもとに，マージコンフリクトの解消方法を予測する機械学習モデルを提案した [17]．白木らの実験においても，片側採用はデータセットに多く含まれていることが示されていた．

マージコンフリクトが片側採用により解消されるということは，並行して進められた開発成果のうち片側が切り捨てられるということでもある．これまでの研究では，マージコンフリクトの解消によるプロジェクトへの影響は多く議論されていないため，開発成果が本当に切り捨てられてよいものかなどの分析は行われていない．

本研究では，マージコンフリクトの解消時に片側採用が選択された理由を調査し，分類を行う．具体的には，Java で書かれた OSS プロジェクトの中から片側採用により解消された

マージコンフリクトを収集し，解消時に削除されたコードの内容や対となるブランチとの差異，コミットメッセージなどをもとに，片側採用の理由を分類する．

以降，2 章では本研究の背景について述べる．3 章では調査の手順と，本研究での片側採用が選択された理由の分類について説明する．4 章で調査結果を説明し，5 章でまとめを述べる．

2　背景

2.1　バージョン管理システムを用いた開発

現代のソフトウェア開発においては，複数の開発者が並行して開発を進めることが多くある．同時並行的な開発作業を支援するツールとして，バージョン管理システム（Version Control System．以下 VCS）が頻繁に用いられる．代表的な VCS として，Git[1]や Apache Subversion[2]が挙げられる．多くの VCS では，ソフトウェアの過去の版を記録しておく機能のほか，複数人が並行して開発した成果を統合する作業を支援する仕組みが取り入れられている．例えば，開発者はまず本流のブランチから新たに自身が作業するための派生ブランチを作成し，派生ブランチ上で編集作業を行う．成果物が完成したら，派生ブランチの作業内容を本流のブランチに統合する．この統合作業をマージという．派生ブランチは同時に複数作成することができるため，複数の開発者が同時に作業を行ったとしても，マージを繰り返すことで効率的にソフトウェア開発を進めることができる．

2.2　マージコンフリクト

VCS における並行開発において，複数のブランチにおける機能の変更内容を統合するマージ作業が行われる．2 つのブランチにおける最新のコミットのマージにおいて，それらのコミットと共通の祖先の差分を用いてマージを行う 3-way マージが一般的である．3-way マージの利点として，共通の祖先に存在する行に対する変更や削除などの編集を行った場合にその編集を検出し，適切なマージを行うことが挙げられる [9]．しかし，2 つのブランチにおける最新のコミットの差分において同一箇所に対する一貫性のない編集が検出された場合，マージコンフリクトが発生する．

マージコンフリクトの例を示す．以下に示すような共通の祖先コミットがあると仮定する．

```
1    int num = 0;
```

このコミットからブランチが作成され，片方のブランチでは，++ 演算子によってインクリメントが行われた操作が追加され，もう片方のブランチでは 1 の加算によってインクリメントが行われた操作が追加された．

```
1    int num = 0;
2  +  num++;
```

```
1    int num = 0;
2  +  num = num + 1;
```

[1]https://git-scm.com/
[2]https://subversion.apache.org/

この際にマージを行うと，それぞれのブランチのコミットの 2 行目において異なる編集が行われている，つまり同一箇所に対する一貫性のない編集が検出されるため，以下のようにマージコンフリクトが発生する．

```
1   int num = 0;
2  <<<<<<<
3   + num++;
4  ++======
5  +  num = num + 1;
6   >>>>>>>
```

Mahmood らの研究によると，編集を行った作業の内容が，リファクタリングまたは機能の追加である場合にマージコンフリクトが発生しやすいという結果がある [10]．リファクタリングや機能の追加は，ソフトウェア開発において，日常的に行われている作業であるため，マージコンフリクトという問題に，多くの開発者が直面している．実際に，開発者たちに調査を行ったところ，回答者の 54%が，マージを行う上で最も重要な問題はマージコンフリクトであると回答している [14]．

2.3　マージコンフリクトの種類

マージコンフリクトは，テキストのコンフリクトとより高次のコンフリクトに分類することが出来る [5]．テキストのコンフリクトとは，2.2 節で述べたような二つのブランチにおけるソースコードの同一箇所に対する一貫性のない編集が原因となり発生したコンフリクトである．このコンフリクトは，VCS によるブランチのマージ操作を行った際に検出することができる．

高次のコンフリクトとは，ビルドのコンフリクトやテストのコンフリクトなどのプログラム実行時に異常が発覚するものを指す [4, 5, 16]．これは意味的に一貫性のない編集が行われたときに発生する．例えば，同一変数に対する操作がそれぞれのブランチでソースコードの異なる箇所で追加され，両ブランチにてテストが通過していた場合を考える．この際にマージ操作を行うと，同一箇所の編集ではないためテキストのコンフリクトは発生しないが，同一変数に対する操作が重複して行われる．テストが通過しない場合，両ブランチでの機能を正常に統合することが出来ていないため，テストコンフリクトが発生したこととなる．高次のコンフリクトは VCS によるブランチのマージ操作を行った際に検出することができない．Pastore らは，ソフトウェアのモデルを作成し各ブランチにおけるソースコードの振舞いを比較することで，高次のコンフリクトを検出する手法を提案している [13]．

Brun らの調査によるとマージ全体においてテキストのコンフリクトが占める割合が 16%であるのに対し，ビルドコンフリクトが占める割合が 6%，テストコンフリクトが占める割

合が 1%であり，テキストコンフリクトが占める割合が高い．そのため，本研究で扱うマージコンフリクトはテキストのコンフリクトとし，これ以降，マージコンフリクトはテキストのコンフリクトを指すものとする．

2.4 マージコンフリクトの探索

マージコンフリクトが発生した場合には，その時点ではマージの結果が VCS には記録されず，マージをしようとした開発者に対してマージコンフリクトを解消するよう促すメッセージが表示される．開発者はその後，コンフリクトを解消した状態にして，改めてコミットすることでマージコミットが作成される．そのため，開発過程で発生したマージコンフリクトの情報は Git に残されていない．開発履歴からマージコンフリクトの調査を行う場合，マージ直前のコミットだけではなく，それ以前の開発履歴を調査して，実際にマージコンフリクトが発生したと考えられる地点を割り出す必要がある．

湯月らは，マージコンフリクトの解消方法を調査するにあたり，マージした時点から開発履歴をさかのぼってブランチ間の距離が極大となる地点を実際にマージを行おうとした地点であると定めた [18]．本研究でも，湯月らの定義を採用して調査を行う．

2.5 マージコンフリクトの軽減

2.2 節で述べたように，マージコンフリクトは同一箇所に対する一貫性のない編集が原因で発生し，マージを行う上で大きな問題となっている．そのため，この問題を解決するためにマージコンフリクトを軽減するための研究がこれまでに多く行われている．Mens の調査によると，マージ時に編集内容に対して操作を加えることで，マージコンフリクトの回避を試みるものが多い [12]．

Semi-structured merge は，構文木の情報を用いて構造的なマージを行う手法である [1, 2, 3]．この手法はソースコードを構文木に変換した後に，その構文木に対してマージ作業を行う．マージ対象のコミットにおいて構文木上の編集に矛盾がなければ，テキストのコンフリクトと判定されるような場合においてもマージを行うことができる．構文木を用いた他の手法として，4-way 差分アルゴリズムという手法を用いたマージも提案されている [15]．4-way 差分アルゴリズムにおいては，マージを行うコミットペアにおいて共通の祖先から変更が行われなかった箇所を差分として用いる．また，マージコンフリクトの解消コストを軽減する方法として，マージコンフリクトの解消に必要な知識を持つ開発者を推薦する TIPMerge といったツールも提案されている [6]．このツールは，開発者のプロジェクトにおける過去の開発経験やブランチでの変更，ブランチ内で修正されたファイルの依存関係などをもとにマージを行うべき開発者の推薦を行っている．マージコンフリクトの軽減において，その変更の

影響の局所化を支援する手法も提案されている．Jackson らは，手続きにおいて二つのバージョン間の差分に着目しその内容を要約するツールである Semantic Diff を開発した [7]．ただし，このツールは手続き内部の処理を比較しているが，手続き間の処理に与える影響に関しては考慮していない．

いずれの手法においてもヒューリスティックにマージコンフリクトの軽減を実施・支援しているため，適切なマージが行われない場合にマージ後に 2.3 節で述べたようなテストやビルドが失敗する高次のコンフリクトが発生してしまう可能性がある．

2.6　マージコンフリクトの解消方法とその予測

いくつかの既存研究において，マージコンフリクトの解消方法に関する調査が行われている．開発においてマージコンフリクトが発生した場合，開発者は経験則に基づいてマージコンフリクトの解消方法を決定し，その解消を行っていることが明らかになっている [11]．また，具体的なマージコンフリクトの解消方法に関して，2 つの親コミットのどちらか片方の編集を採用し，もう片方の親の編集を削除することで，マージコンフリクトの解消が行われるケースが多く存在していることが知られている．これを片側採用といい，湯月らの研究によって，Java プロジェクト中のメソッド内部で発生したマージコンフリクトにおいては，全体の約 98% が片側採用で解消されていることが明らかになっている [18]．

白木らは，OSS におけるマージコンフリクトの解消方法について，機械学習を用いてその解消方法を予測する手法を提案した．白木らのモデルは，マージ元となるそれぞれのブランチに対して表 1 のいずれかの分類を付与し，その組み合わせを最終的な予測結果として出力する．例えば，片方のブランチの編集内容が全てマージ後に採用されており，もう片方のブランチの編集内容は一部分のみが採用された場合の分類は [ADOPT, EDIT] となる．予測に用いる情報はすべて Git リポジトリからコマンドを用いて取得可能な開発履歴のメタ情報であり，アルゴリズムにはランダムフォレストを採用した．Apache ソフトウェア財団が公開している OSS の中から，Java プロジェクト 20 個を対象に実験を行った結果，平均で約 66%程度の正答率を得た．

表 1: 白木らによるマージコンフリクト解消方法の分類 [17]．これらの分類はそれぞれのブランチに対して付与され，その組み合わせが実際に出力される．

ADOPT	編集内容を全て採用する
DELETE	編集内容を全て消去する
EDIT	編集内容の一部を採用する，追加の編集を行う，またはその両方
ZERO	編集箇所が 0 行である

白木らの実験では，正答率が高いプロジェクトで特に実際のマージコンフリクトの解消方法に偏りがあったことも報告されており，正答率約 90%のプロジェクトにおいては [DELETE, ADOPT] の片側採用が 6 割程度を占めていた．

3　調査手順

本研究では，OSS においてマージコンフリクト解消時に片側採用が選択されることが多いことにもとづき，なぜそのような選択がなされたのか理由の調査を行う．まず，マージコンフリクトが発生したプロジェクトの中で片側採用が選択されたコミットを調査する．その後，片側採用が選択されたマージコンフリクトについて，マージ前のそれぞれのブランチの内容を比較し，採用された側と削除された側の形式的な違いで分類する．

3.1　調査対象プロジェクト

本研究では，白木らの既存研究 [17] で対象とされた Apache プロジェクトが提供する OSS である 20 個の Java プロジェクトの内，7 個のプロジェクトを対象として調査を行う．Java プロジェクトを対象とするため，コンフリクトが Java ファイル以外の xml ファイルや py ファイルで発生しているものは今回の調査の対象外とする．Java ファイルにおいてコンフリクトが発生したものに関しても，テストケースにおいてコンフリクトが発生しているものは対象から除外した．また，片側採用が利用されているものの，両方のブランチから一部分ずつを採用することによってコンフリクトを解消しているものや，片側採用の箇所と編集によって解消した箇所が混在しているものも，その理由を明確にすることが困難であるため，本研究では調査対象外とした．具体的には，白木らの研究における解消方法のうち [ADOPT, DELETE] となるもののみについて調査を行い，それ以外の片側採用である [EDIT, DELETE] や [ZERO, DELETE] については対象外とした．

対象とするプロジェクトに含まれていたマージコンフリクトの数と，その中で片側採用が選択されたものの内上記のものを除いた数は表 2 の通りである．この中で，beam，cordova-android，dubbo の 3 つのプロジェクトに対して更に調査を行う．片側採用が選択されたコミットに対して，なぜそのような選択されたのかの理由を分類を行う．

表 2: マージコンフリクトの数

	マージコンフリクトの数	片側採用のみで解消された数
beam	1,146	494
cordva-android	700	70
dubbo	1,010	248
geode	1,790	345
jmeter	395	303
nifi	620	15
nutch	445	180

3.2 分類方法

片側採用が選択された理由の分類を行う手法を述べる．以下のコマンドを実行し，2 つの親コミット P1，P2 を持ち，ファイル F で発生したマージコンフリクトにおいて，そのテキストの差分を得ることが出来る．

```
git diff P1 P2 F
```

このコマンドによって得られた差分から，筆者が目視でその変更内容を精査し，以下に述べる分類のどれに当てはまるかを判断する．

- 新規機能の実装
- 機能の削除
- 異なる機能の追加
- フォーマット
- コメント
- import 文
- リファクタリング
- その他

3.3 分類の内訳

マージコンフリクトの解消として片側採用が選択された理由の分類に関して，その分類内容と判断基準を述べる．

3.3.1 新規機能の実装

片側採用が選択される際，新規に機能を実装するときにマージコンフリクトが発生し，その新規機能を実装したコミットを採用したものを新規機能の実装として分類する．

テキスト上では，採用されなかった方のソースコードには何も書かれておらず採用された方のソースコードにのみプログラムが書かれているものや，採用されなかった方のソースコードにもプログラムが書かれているが，採用されたソースコードに含まれる内容しか書かれていないものをこの分類にしている．

3.3.2 機能の削除

新規機能の実装とは逆に，機能が実装されたコミットと実装されていないコミットの間でマージコンフリクトが発生し，機能が実装されていないコミットを採用したものを機能の削除として分類する．

採用された方のソースコードにプログラムが書かれていないものや，採用されたソースコードにプログラムが書かれているが，採用されなかった方のソースコードに含まれる内容しか書かれていないものをこの分類にしている．

3.3.3 異なる機能の追加

2 つのブランチで実装されている機能が異なっている状態でマージコンフリクトが発生し，その解消のため片方の機能を削除し，片方の機能を採用する形で片側採用を選択したものを異なる機能の追加として分類する．

採用された方のソースコードと採用されなかった方のソースコードの間でテキスト上で明らかな違いがあり，その機能に差が生じていると考えられるものをこの分類にしている．実際の開発において，採用された方のソースコードが正しい挙動を示しているかはここでは考えておらず，単にソースコードの内容の差で判断をしている．

3.3.4 フォーマット

マージコンフリクトが発生しているものの，その機能に差がなく，インデントや改行などのフォーマット上での違いしか無いコンフリクトで片側採用が選択されたものをフォーマットとして分類する．

3.3.5 コメント

プログラム本体ではなく，コメントアウトされている箇所でマージコンフリクトが発生しており，プログラムの機能に影響が無い範囲で片側採用が選択されたものをコメントとして分類する．

コメントの変更には，ソースコード上である機能が変更された際にそれに付随するコメントがあわせて変更される場合もある．今回の調査では，そのような場合はソースコードの変更に応じた他の分類に振り分け，ライセンスを記載したものやファイルの説明をしているもの，開発上のメモを残しているだけのものといったコメントそのもので役割が完結している部分で片側採用が選択されたものをこの分類にしている．

3.3.6 import 文

ソースコードの変更に伴って，import 文の部分で変更があるためにマージコンフリクトが発生し，その解消のため採用されたソースコードに合わせて import 文で片側採用が行われたものを import 文として分類する．

3.3.7 リファクタリング

リファクタリングの際にマージコンフリクトが発生し，その解消のために片側採用が選択されたものをリファクタリングとして分類する．

テキスト上では 2 つのブランチで書かれていることが異なっているものの，その内容がメソッドの抽出や変数名の変更などであり，機能自体に変化が無いと判断されたものをこの分類にしている．

3.3.8 その他

マージコンフリクトの解消に片側採用が選択されているものの，その理由が上記に当てはまらないものなどをその他として分類する．

その他の内訳としては，マージコンフリクトが発生しているもののテキスト上での差分が確認できなかったものや，デバッグの痕跡である print 文を消去したもの，コンフリクトの発生した個所や行数が特定できなかったために分類することが不可能だったものなどがある．

4 調査結果

4.1 分類の結果

前章の内容に基づいて調査した結果を表 3 に示す.

beam と dubbo については，フォーマットの割合が最も高く，次いでコメント，import 文の割合が高いという結果になった．cordva-android については，フォーマットの割合が最も高く，次いで機能の削除，新規機能の実装と異なる機能の追加の割合が高いという結果になった．サンプル数が少なかった cordva-android については他の 2 つと傾向に差が出たものの，全体としてフォーマットに起因するマージコンフリクトが多くを占めていることがわかった.

4.2 各分類の具体例

ここでは，今回の調査で見つかった各分類の具体例を示す.

4.2.1 新規機能の実装

図 1 は，beam プロジェクト内の DoFnSignature.java で起きた片側採用の事例である．このパターンでは，ブランチ 1 が採用された．採用されたソースコードでは，採用されなかった方のソースコードに存在しない機能が追加されている.

4.2.2 フォーマット

図 2 は，cordova-android プロジェクト内の CordovaChromeClient.java で起きた片側採用の事例である．このパターンでは，ブランチ 1 が採用された．2 つのソースコードで書かれ

表 3: 片側採用の選択理由

	beam	cordva-android	dubbo
新規機能の実装	7.4%	8.3%	6.6%
機能の削除	3.7%	18.3%	3.3%
異なる機能の追加	7.6%	8.3%	3.3%
フォーマット	41.6%	48.3%	45.5%
コメント	19.2%	1.7%	13.2%
import 文	10.8%	1.7%	10.4%
リファクタリング	2.9%	1.7%	6.1%
その他	6.8%	11.7%	11.3%

ている内容には，インデントがずれている以外に違いがない．

4.2.3 リファクタリング

図 3 は，dubbo プロジェクト内の ExtensionLoader.java で起きた片側採用の事例である．このパターンでは，ブランチ 1 が採用された．どちらも同じ機能を実装しているが，文字列の処理を演算子で行うか apend で行うかで異なっている．

4.3 考察

本研究の調査により，マージコンフリクトが発生し片側採用が選択されることで解決されたものの中では，フォーマットに関するものの割合が高いことが判明した．プロジェクトで整形ルールを決めてマージ前に自動的に適用することで，マージコンフリクトの発生件数を大幅に減らすことが可能になると考えられる．

比較的割合の高かったコメントに関してだが，自動プログラム修正など，ソースコードの挙動から正しく動作するプログラムを作成するアプローチはこのパターンでは参考にすることが出来ず，このケースのコンフリクトには別個に検討する必要があるだろう．

新規機能の実装，機能の削除，異なる機能の追加といったケースに関しては，ツールを用いて修正作業を行う前に，どの機能を採用してどの機能を残すかの判断が必要になる．その判断に応じてテストケースを正しく準備することができれば，解消自体は自動化ツールの恩恵を受けられる可能性がある．

また，機能の削除や異なる機能の追加については，製作者の意図する通りにコンフリクトが解消されていない可能性がある．他の部分のコンフリクト解消の際に誤って機能を削除してしまうケースや異なる機能を組み合わせることで本来必要とされる機能を実装できるケースなどが考えられるためである．この件に関して，機能が削除されたものを含むコンフリクトに関しては，その解消後同じような機能が改めて実装されてないかといった追跡調査が必要になると考えられる．

ブランチ１（採用）

```
+  /**
+   * Describes a state declaration; a field of type {@link StateSpec}
annotated with
+   * {@link DoFn.StateId}.
+   */
+  @AutoValue
+  public abstract static class StateDeclaration {
+    public abstract String id();
+    public abstract Field field();
+    public abstract TypeDescriptor<? extends State> stateType();
+
+    static StateDeclaration create(
+        String id, Field field, TypeDescriptor<? extends State> stateType) {
+      return new AutoValue_DoFnSignature_StateDeclaration(id, field,
stateType);
+    }
+  }
+
   /** Describes a {@link DoFn.Setup} or {@link DoFn.Teardown} method. */
   @AutoValue
+  public abstract static class LifecycleMethod implements DoFnMethod {
+    /** The annotated method itself. */
+    @Override
    public abstract Method targetMethod();

    static LifecycleMethod create(Method targetMethod) {
      return new AutoValue_DoFnSignature_LifecycleMethod(targetMethod);
    }
  }
```

ブランチ２（不採用）

```
/** Describes a {@link DoFn.Setup} or {@link DoFn.Teardown} method. */
   @AutoValue
-  public abstract static class LifecycleMethod {
public abstract Method targetMethod();

    static LifecycleMethod create(Method targetMethod) {
      return new AutoValue_DoFnSignature_LifecycleMethod(targetMethod);
    }
  }
```

図 1: 「新規機能の実装」に分類された片側採用の例

ブランチ１（採用）

```
 dlg.setPositiveButton(android.R.string.ok,
+                new DialogInterface.OnClickListener() {
+                    public void onClick(DialogInterface dialog, int which) {
+                        result.confirm();
+                    }
+                });
dlg.setNegativeButton(android.R.string.cancel,
+                new DialogInterface.OnClickListener() {
+                    public void onClick(DialogInterface dialog, int which) {
+                        result.cancel();
+                    }
+                });
```

ブランチ２（不採用）

```
 dlg.setPositiveButton(android.R.string.ok,
-            new DialogInterface.OnClickListener() {
-                public void onClick(DialogInterface dialog, int which) {
-                    result.confirm();
-                }
-            });
         dlg.setNegativeButton(android.R.string.cancel,
-            new DialogInterface.OnClickListener() {
-                public void onClick(DialogInterface dialog, int which) {
-                    result.cancel();
-                }
-            });
```

図 2: 「フォーマット」に分類された片側採用の例

ブランチ１（採用）

```
public class ExtensionLoader<T> {
                        type.getName(), Arrays.toString(value));
                code.append(s);

+               s = String.format("¥n%s extension = null;¥n try {¥nextension =
(%<s)%s.getExtensionLoader(%s.class).getExtension(extName);¥n}catch(Exception e){¥n",
+                       type.getName(), ExtensionLoader.class.getSimpleName(), type.getName());
+               s += String.format("if (count.incrementAndGet() == 1) {¥nlogger.warn(¥"Failed to find extension
named ¥" + extName + ¥" for type %s, will use default extension %s instead.¥", e);¥n}¥n",
+                       type.getName(), defaultExtName);
+               s += String.format("extension = (%s)%s.getExtensionLoader(%s.class).getExtension(¥"%s¥");¥n}",
+                       type.getName(), ExtensionLoader.class.getSimpleName(), type.getName(), defaultExtName);
+               code.append(s);
```

ブランチ２（不採用）

```
public class ExtensionLoader<T> {
                        type.getName(), Arrays.toString(value));
                code.append(s);

-               code.append(String.format("¥n%s extension = null;¥n try {¥nextension =
(%<s)%s.getExtensionLoader(%s.class).getExtension(extName);¥n}catch(Exception e){¥n",
-                       type.getName(), ExtensionLoader.class.getSimpleName(), type.getName()));
-               code.append(String.format("if (count.incrementAndGet() == 1) {¥nlogger.warn(¥"Failed to find
extension named ¥" + extName + ¥" for type %s, will use default extension %s instead.¥", e);¥n}¥n",

-                       type.getName(), defaultExtName));
-               code.append(String.format("extension =
(%s)%s.getExtensionLoader(%s.class).getExtension(¥"%s¥");¥n}",
-                       type.getName(), ExtensionLoader.class.getSimpleName(), type.getName(),
defaultExtName));
```

図 3: 「リファクタリング」に分類された片側採用の例

5　まとめ

本研究では，マージコンフリクトの解消に片側採用が選択された際の理由について調査した．片側採用が選択されたマージコンフリクトについて，マージ前のそれぞれのブランチの内容を比較し，形式的な違いで 8 種類に分類した．調査の結果，片側採用が選択されたマージコンフリクトでは，フォーマットやコメント，import 文に関するものが多いことが分かった．また，調査の過程で得られた情報から，整形ルールを決めてマージ前に適用することで，マージコンフリクトの削減を行うなど，マージコンフリクトの削減やマージコンフリクトの解消時の負担軽減に繋がる手法を考察した．

今後の課題として，二つのブランチから一部分ずつを片側採用することでコンフリクトを解消したものや，二つのブランチの内容を組み合わせて編集することでコンフリクトを解消したものの数が多く残っていることがある．それらについての分析はかなり難しくなるが，同様の調査を行うことで，開発者の支援にさらに近づくことができると考えられる．

謝辞

大阪大学大学院情報科学研究科コンピュータサイエンス専攻井上克郎教授には，研究において多くの御指導及び御助言を賜りました．井上教授の御指導のおかげで，研究を進め本論文の完成に至ることができました．井上教授に心より感謝いたします．

大阪大学大学院情報科学研究科コンピュータサイエンス専攻松下誠准教授には，研究の各段階において適切な御助言を賜りました．多くの御助言を頂いた松下准教授に心より感謝いたします．

大阪大学大学院情報科学研究科コンピュータサイエンス専攻神田哲也助教には，研究において大変多くの御指導及び御助言を賜りました．また，研究に関する数多くの相談に乗っていただきました．神田助教に心より感謝いたします．

大阪大学大学院情報科学研究科コンピュータサイエンス専攻嶋利一真氏には，研究に関する相談に乗っていただきました．特に論文執筆において，数多くの御指導及び御助言を賜りました．嶋利一真氏に心より感謝いたします．

最後に，大阪大学大学院情報科学研究科コンピュータサイエンス専攻井上研究室の皆様には，多くの御指導及び御助言を頂きました．井上研究室の皆さまに心より感謝いたします．

参考文献

[1] Paola Accioly, Paulo Borba, and Guilherme Cavalcanti. Understanding semi-structured merge conflict characteristics in open-source Java projects. Vol. 23, p. 2051–2085, 2018.

[2] Sven Apel, Olaf Leßenich, and Christian Lengauer. Structured merge with auto-tuning: balancing precision and performance. In *International Conference Automated Software Engineering(ASE)*, pp. 120–129, 2012.

[3] Sven Apel, Jörg Liebig, Benjamin Brandl, and Christian Kästner Christian Lengauer. Semistructured Merge:Rethinking Merge in Revision Control Systems. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pp. 190–200, 2011.

[4] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. Proactive Detection of Collaboration Conflicts. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, p. 168–178, 2011.

[5] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. Early Detection of Collaboration Conflicts and Risks. *IEEE Transactions on Software Engineering*, Vol. 39, No. 10, pp. 1358–1374, 2013.

[6] Catarina Costa, Jair Figueiredo, Leonardo Murta, and Anita Sarma. TIPMerge: Recommending Experts for Integrating Changes across Branches. In *Fast Software Encryption(FSE)*, pp. 523–534, 2016.

[7] Daniel Jackson and David A. Ladd. Semantic diff: a tool for summarizing the effects of modifications. In *Proceedings 1994 International Conference on Software Maintenance (ICSM)*, pp. 243–252, 1994.

[8] Bakhtiar Khan Kasi and Anita Sarma. Cassandra: Proactive Conflict Minimization through Optimized Task Scheduling. In *International Conference on Software Engineering(ICSE)*, pp. 732–741, 2017.

[9] Ernst Lippe. Operation-based Merging. In *the fifth ACM SIGSOFT symposium on Software development environments(SDE5)*, pp. 78–87, 1992.

[10] Wardah Mahmood, Moses Chagama, Thorsten Berger, and Regina Hebig. Causes of Merge Conflicts: A Case Study of ElasticSearch. In *International Working Conference on Variability Modelling of Software-Intensive Systems*, 2020.

[11] Shane McKee, Nicholas Nelson, Anita Sarma, and Danny Dig. Software Practitioner Perspectives on Merge Conflicts and Resolutions. In *International Conference on Software Maintenance and Evolution (ICSME)*, pp. 467–478, 2017.

[12] Tom Mens. A State-of-the-Art Survey on Software Merging. No. 5, pp. 449–462, 2002.

[13] Fabrizio Pastore, Leonardo Mariani, and Daniela Micucci. BDCI: Behavioral Driven Conflict Identification. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pp. 570–581, 2017.

[14] Shaun Phillips, Jonathan Sillito, and Rob Walker. Branching and Merging:An Investigation into Current Version Control Practices. In *the 4th International Workshop on Cooperative and Human Aspects of Software Engineering(CHASE)*, pp. 9–15, 2011.

[15] MARCELO SOUSA, ISIL DILLIG, and SHUVENDU K. LAHIRI. Verified Three-Way Program Merge. *Proceedings of ACM on Programming Languages*, Vol. 2, No. 165, pp. 1–29, 2018.

[16] Jae young Bang, Daniel Popescu, and Nenad Medvidovic. Enabling Workspace Awareness for Collaborative Software Modeling. In *ACM Conference on Computer Supported Cooperative Work(CSCW)*, 2012.

[17] 白木秀弥, 神田哲也, 井上克郎. 機械学習による開発履歴のメタ情報を用いたマージコンフリクトの解消パターン判定モデル. 信学技報, 第 119 巻, pp. 61–66, 3 2020.

[18] 湯月亮平. 開発履歴を用いたメソッドコンフリクトの分析. 修士論文, 奈良先端科学技術大学院大学, 2015.