情報処理学会論文誌 Vol.62 No.12 2041–2055 (Dec. 2021)

自動プログラム修正を用いたマージ競合の解決

丸山 勝久[1,a)] 邢 小茜[2]

受付日 2021年3月18日, 採録日 2021年9月9日

概要：異なる開発者によってソースコードが独立に編集可能な並行開発において，マージ競合は避けられない．競合を解決するためには，マージ後のソースコードに対して期待される振舞いを検査するすべてのテストに成功するように，競合するコード片を取り除かなければならない．この作業は，ソースコードに対する深い理解が必要であり，開発者にとって面倒な作業である．本論文では，従来手法ではいまだ取り組まれていない振舞い競合を自動解決する新たな手法を提案する．この手法では，マージ対象の Java ソースコード内部に存在するクラスメンバを組み合わせることで，コンパイル可能でバグを含む（可能性のある）ソースプログラムを生成する．その後，自動プログラム修正技法によりバグを取り除くことで，マージ競合を含まないソースプログラムを出力する．このような 2 段階の手順を明確に規定することで，振舞い競合の解決に自動プログラム修正技法を活用することがはじめて可能となった．さらに，提案手法を用いた 3 つの実験において，受け入れ可能なマージ後のソースプログラムの出力に成功した．このことより，提案手法が人手の介入を抑えて振舞い競合を解決できることを立証した．

キーワード：並行開発，マージ競合，自動プログラム修正，遺伝的アルゴリズム

Resolving Merge Conflicts Using Automated Program Repair

Katsuhisa Maruyama[1,a)] Xiaoqian Xing[2]

Received: March 18, 2021, Accepted: September 9, 2021

Abstract: Merge conflicts are inevitable in concurrent software development where source code has been independently modified by multiple programmers. Unfortunately, the resolution of such merge conflicts is troublesome for programmers since they have to remove the conflicting code fragments until the merged code passes all tests that check their expected behavior. In this paper, we propose a novel mechanism that reduces programmers' burden to resolve behavioral merge conflicts which conventional mechanisms do not still address. It produces compilable source programs that might contain faults by combining class members within Java source code to be merged. Then, it outputs source programs not including any merge conflicts by exploiting an automated program repair (APR) technique that fully-automatically fixes faults exposed by tests. Clearly prescribing this two-stage procedure makes it possible to implement the automatic merge mechanism using an APR technique for the first time. In all three experiments with the mechanism, merged source programs acceptable to programmers were successfully obtained. These experimental results demonstrated that the mechanism can solve behavioral conflicts with little intervention of human.

Keywords: concurrent software development, merge conflicts, automated program repair, genetic algorithm

1. はじめに

Git や Mercurial のような版管理システムを利用するソフトウェア開発では，それぞれの開発者が自分のワークスペース（ブランチ）でソースプログラム[*1]の変更を行い，

[1] 立命館大学情報理工学部
Department of Information Science and Engineering, Ritsumeikan University, Kusatsu, Shiga 525–8577, Japan
[2] 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
[a)] maru@cs.ritsumei.ac.jp

[*1] 本論文では，ソースコードのインスタンスを扱うため，ソースコードのことをソースプログラムと呼ぶ．

変更の完了後にそれらをマージする．このような並行開発では，複数の開発者が独立して編集作業を行うことが可能となり，個々の変更が効率的に実施できる [1], [2].

その一方で，並行開発が必ずしもソフトウェア開発の生産性を高めるとは限らない [3]．それぞれの開発者は，自分以外のワークスペースで行われた変更を意識して，自分のソースプログラムを変更しているわけではない．このため，複数の開発者により独立に変更されたソースプログラムの断片が，もとのソースプログラムにおいて重なったり，互いに干渉したりすることがある [3]．この重なりや干渉は，ソースプログラムをマージする際に表面化する．これを，マージ競合（merge conflict）という [4]．マージ競合の存在が並行開発を推進するうえでの障壁になっている [5], [6], [7], [8].

マージ競合は大きく，マージ途中のソースプログラムが，(1) 構文解析に失敗する構文的競合，(2) コンパイルに失敗する静的な意味的競合（ビルド競合），(3) 予期しない振舞いを実行する振舞い競合（テスト競合）に分類される [4]．ここで，開発者が手動でマージした，あるいはツールにより自動的にマージされたソースプログラムに残された構文競合や静的な意味的競合は，コンパイラを利用することで検出することができる．さらに，コンパイルが成功したソースプログラムに対して十分なテストを実施することで，振舞い競合を検出することも可能である [9], [10].

しかしながら，コンパイルやテストによって競合が検出できたとしても，それらはマージ競合を解決するわけではない．マージ競合の解決とは，競合する 2 つの変更の一部を取り出して，それらを矛盾なく（重なりや干渉をなくして）混ぜ合わせることを指す．この作業は，開発者がソースプログラムの差分やマージ途中のソースプログラムを見ながら注意深く行う必要があり，開発者にとって大きな負担となっている [11], [12], [13].

本論文では，マージ対象のソースプログラムに対して，自動プログラム修正（以下 APR と呼ぶ）技法 [14] を活用して，振舞い競合を自動的に解決する手法を提案する．APR とは，バグ（欠陥）を含むプログラムとその振舞いに関する正解を与えて，バグを除去するパッチ（修正前後のソースコードの差分）を出力する技術である．

提案手法では，マージ対象の Java ソースプログラム内部に存在するクラスメンバを組み合わせることで，コンパイル可能な複数のソースプログラムを最初に作成する．本論文では，このように作成した個々のソースプログラムを初期ソースプログラムと呼ぶ．初期ソースプログラムは，互いの変更における重なりが解消されるものの，一般的にマージ後のソースプログラムに求められる振舞いを満たさない．この場合，その初期ソースプログラム中にバグが存在していると見なす．APR によりバグを取り除く際には，変異個体を生成する際の修正材料として，マージ対象のソースプログラムの一部を利用する．これにより，マージ対象のソースプログラムにおける互いの変更の混ぜ合わせを実現する．バグが除去されたソースプログラムはマージ後に求められる振舞いを満たすため，振舞い競合を解決したソースプログラムが獲得できる．

我々の知る限り，本手法がマージ競合の解決に対する APR の最初の応用である [15]．本論文の貢献を以下に示す．

- 従来手法では対象としていない振舞い競合の自動解決を目指して，APR を活用したマージ手法 APR-MR（APR-based Merge Resolution）を提案する．
- マージ対象のソースプログラムから修正対象のソースプログラムを機械的に作成することで，APR-MR の自動化を実現する仕組みを示す．
- 3 つの実験を通して，APR-MR により振舞い競合の自動解決が可能なことを立証する．

以下，2 章ではマージ競合の例を示す．3 章では APR を活用した APR-MR を説明する．4 章では実験の内容とその結果を示す．5 章で関連研究を紹介する．6 章でまとめを述べる．

2. マージ競合とその解決の例

開発者 A と B が独立してソースプログラムを編集することで，マージ競合が発生する例を**図 1** に示す．ここでは，クラス `Book` を定義するソースプログラムを Book.java としている．もとのソースプログラム S_O から，それぞれの開発者が作成したソースプログラムを S_A と S_B とする．S_A および S_B において，赤色あるいは緑色の網掛けコードがそれぞれの開発者の改変部分である．T_A と T_B は，それぞれ S_A と S_B を検査するテストケースである．この例では，Git による 3-way マージに失敗し，図 1 の右側に示すような競合が報告される（メッセージの一部を改行している）．これは，開発者 A と B の編集が，クラス `Book` のコンストラクタ `Book()` で重なっていることを示している．

このような競合の解決において，マージが成功したかどうかを検査するためには，マージ後のソースプログラム（S_M とする）に対してテストケースを用意するのが一般的である．ここでは，開発者の B の視点から，メソッドの `getPrice()` と `getPriceInfo()` の振舞いを検査するテストスイート（テストケースの集合）T_M を用意した．**図 2** に示すように，T_M は 6 個のテストケースを持つ．さらに，T_M を満たす S_M の候補として我々が事前に用意した，2 つのソースプログラム $S_{M\text{-}A1}$ と $S_{M\text{-}A2}$ を図 2 に示す．

ここで，図 2 に示す 6 個のテストケースには，`assert` 文の実行順序が異なるだけのもの（たとえば，メソッド `testGenPrice1()` と `testGenPriceInfo1()`）が存在する．マージによるソースプログラムの書き換え範囲が限定されているのであれば，このような冗長にみえるテストケースをなくすことは可能である．たとえば，マージによる書き換

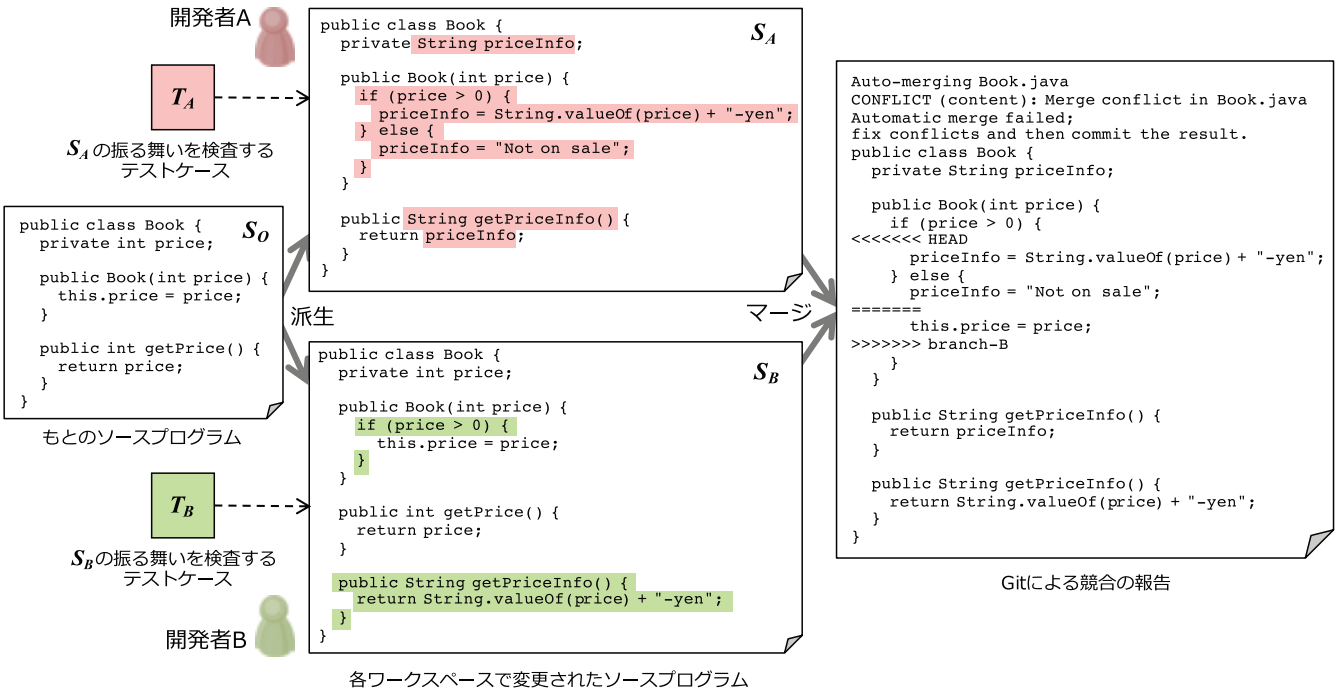

図 1 マージ競合の例

Fig. 1 Example of a merge conflict.

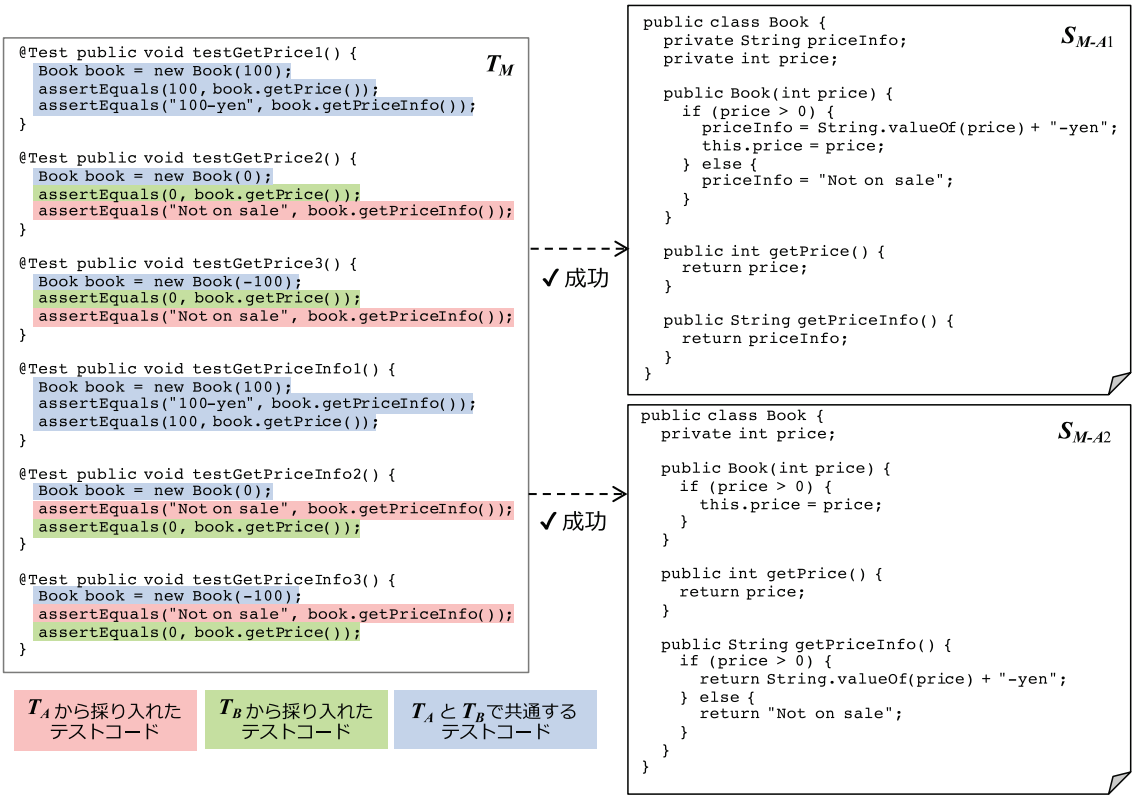

図 2 マージ競合を検査するテストスイートと手動マージにより生成したソースプログラム

Fig. 2 Test suites to validate source programs after the merge and manually merged source programs.

えが Book() だけであった場合，T_M の 6 個のテストケースにおける最終行の assert 文は不要である．しかしながら，$S_{M\text{-}A2}$ に示すような getPriceInfo() の書き換えまでを想定した場合，メソッド getPrice() と getPriceInfo() の呼び出し順序により，クラス Book の振舞いが変わらないことを確認するテストケースが必要となる．いま，マージにより，getPriceInfo() において，フィールド price の値を変更する文が誤って挿入された場合を考える．その際，

testGenPriceInfo1() を実行することで，この誤りが検出できる．これに対して，getPrice() と getPriceInfo() を単独に呼び出すテストケースだけでは，この誤りを検出できない．ここでは，マージ前にテストケースを作成することを前提としているため，一見冗長にみえるテストケースが用意されている．

S_A および S_B において行われた変更の範囲が単一のソースファイルに限定されていたとしても，競合を解決するソースプログラムはいくつも存在する．さまざまな解決方法を確認することが理想的であるが，マージ後のソースプログラムをいくつも作成する手間は開発者にとって負担である．また，実際の並行開発における編集は複数のソースファイルにまたがることが多い．この場合，人手による振舞い競合の解決はさらに面倒になる．

3. APR を利用したマージ競合の解決手法

バグを含むソースプログラムを修正する可能性のあるパッチ候補を（大量に）生成し，生成したパッチにより修正されたソースプログラムが与えられたテストケースを満たすかどうかを検証する APR の戦略に，生成&検証（generate-and-validate）[16] がある．APR-MR では，生成&検証戦略が Java プログラムにおけるさまざまなバグを修正できたという事実 [17] に基づき，この戦略を遺伝的プログラミングで実現した APR システムを採用した．修正後のソースプログラムがすべてのテストケースを通過（テストに成功）した場合，自動バグ修正が成功したと見なす．

遺伝的アルゴリズムに基づく APR では，コンパイル可能な修正対象のソースプログラム，修正後のソースプログラムが満たすテストケース，対象ソースプログラムにおいてコード片を変異させるための修正材料（再利用するコード片の候補）が必要である．APR-MR の利用において，テストケースはあらかじめ用意されていることを前提とした．このようなテストケースは，競合が発生した際に，それぞれの開発者の合意形成に利用され，たとえ APR を活用しない場合にも必要である [18], [19]．

コンパイル可能な修正対象の初期ソースプログラムは，開発者が別途用意するのではなく，マージ対象のソースプログラムから作成する．その際，もとのソースプログラム（図 1 の S_O）およびそれぞれの開発者が作成した 2 つのソースプログラム（図 1 の S_A と S_B）をそのまま初期ソースプログラムとして利用することが考えられる．しかしながら，マージ競合の解決にはフィールド宣言やメソッド定義の追加が必要にもかかわらず，本手法で用いる遺伝的アルゴリズムに基づく APR システムはこのような変異操作を提供していない．このため，初期ソースプログラムについては，APR システムの外部で機械的に作成することにした．

修正材料には，マージ対象のソースプログラムの全部あるいは一部をそのまま利用する．このような方針を採用した理由は，「マージ後のソースプログラムに含まれるコード片は，マージ対象のソースプログラムに含まれている可能性が高い」という著者らの見解に基づいている．この見解は，Ghiotto らの調査結果 [20] に一致する．彼らは，GitHub 上の Java プロジェクトを調査し，マージ後のソースプログラムにおいて，新規に追加されたコード片はほとんど含まれないことを明らかにした．さらに，このような方針を採用することで，開発者にとって見慣れないコード片がマージ後のソースプログラムに突然現れることを避けることができる．

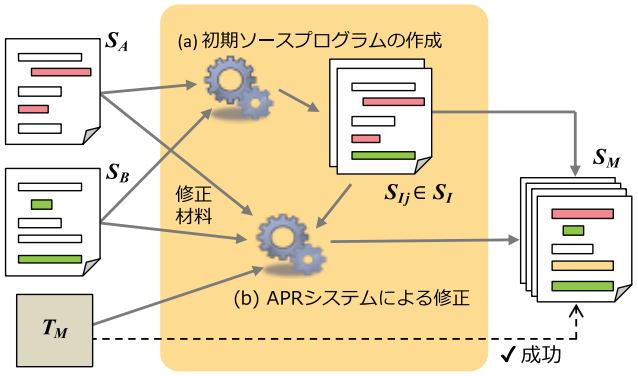

図 3 APR-MR の概要
Fig. 3 Overview architecture of APR-MR.

上記に述べた方針に基づく APR-MR の概要を図 3 に示す．この手法は大きく 2 つのステップ (a)，(b) により，ソースプログラムの振舞いを検査するテストスイート T_M を満たすマージ後のソースプログラム S_M を出力する．

(a) 初期ソースプログラムの作成
マージ対象のソースプログラム S_A と S_B の内部に存在するクラスメンバを組み合わせることによって，修正対象となる初期ソースプログラムの集合 S_I を作成する．

(b) APR システムによる修正
ステップ (a) により作成された初期ソースプログラム S_{Ij}（$\in S_I$）が T_M を満たさない場合，APR システムを実行することで，S_{Ij} を修正する．T_M を満たす S_{Ij} はそのまま S_M とする．

2 つのステップをそれぞれ 3.1 節と 3.2 節で説明する．

3.1 初期ソースプログラムの作成

このステップでは，マージ対象のソースプログラム S_A と S_B に含まれるクラスをメンバ（フィールドとメソッド）に分解し，それらを組み合わせることで，初期ソースプログラムの集合 S_I を生成する．以下，図 1 に示したマージ競合の例を用いて，マージ対象の S_A と S_B から S_I を生成する様子を示す．

ここで，Java のソースコードにおいて，クラスはクラスメンバしか持たない．よって，単一のソースプログラムは

クラスのメンバの集まりで表現できる．いま，ソースプログラム S が，それに含まれるクラスメンバ $m_1, \cdots, m_n$ の集まりで構成されていることを，$S = \langle\, m_1, \cdots, m_n \,\rangle$ と表現する．S_A と S_B は，それぞれ次のように定義できる．

$S_A = \langle\, m_{A1}, m_{A2}, m_{A3} \,\rangle$
$S_B = \langle\, m_{B1}, m_{B2}, m_{B3}, m_{B4} \,\rangle$
m_{A1}: S_A のクラス `Book` のフィールド `priceInfo`
m_{A2}: S_A のクラス `Book` のメソッド `Book()`
m_{A3}: S_A のクラス `Book` のメソッド `getPriceInfo()`
m_{B1}: S_B のクラス `Book` のフィールド `price`
m_{B2}: S_B のクラス `Book` のメソッド `Book()`
m_{B3}: S_B のクラス `Book` のメソッド `getPrice()`
m_{B4}: S_B のクラス `Book` のメソッド `getPriceInfo()`

通常，遺伝的アルゴリズムに基づく APR システムはメソッド本体を修正の対象とし，新たなフィールド宣言やメソッド定義を追加することはしない．さらに，数多くの変異個体がコンパイル可能となるためには，参照される（可能性のある）フィールドや呼び出される（可能性のある）メソッドが初期ソースプログラムに最初から含まれている方がよい．そこで，APR-MR では，できるだけ数多くのクラスメンバを個々の初期ソースプログラムに含ませるという方針をとる．

ここで，Java 言語では，同一のクラスにおいて同一のシグニチャ（フィールドの場合は同一の名前，メソッドの場合は同一の名前と引数の型のリスト）を持つクラスメンバが共存できない．よって，このようなクラスメンバは初期ソースプログラムに排他的に含まれることになる．その際，同一のシグニチャを持つクラスメンバのすべての組合せを考えると，生成される初期ソースプログラムの数が膨大になる可能性がある．これを避けるため，APR-MR では，以下の方針により組合せの数を削減する．

いま，もとのクラス C_O に対して，それを変更したクラス C_A と C_B に含まれるクラスメンバ m_{Ai} と m_{Bj} を考える．m_{Ai} と m_{Bj} は同一のシグネチャを持つ．m_{Ai} と m_{Bj} の両方が書き換えられている（m_{Ai} および m_{Bj} と同じシグネチャを持つ C_O のクラスメンバ m_{Ok} が存在し，m_{Ok}，m_{Ai}，m_{Bj} の本体は互いに異なる）場合，あるいは m_{Ai} と m_{Bj} の両方が新規に追加された（m_{Ai} と m_{Bj} のどちらも C_O に存在しない）場合，m_{Ai} と m_{Bj} を排他的関係と定義する．この関係を $m_{Ai}|m_{Bj}$ と表現する．これに対して，m_{Ai} と m_{Bj} のどちらか一方のみが書き換えられている場合，m_{Ai} と m_{Bj} を排他的関係ととらえず，書き換えられているクラスメンバのみを初期ソースプログラムに挿入する．3-way マージと異なり，m_{Ai} と m_{Bj} のどちらか一方が削除された場合には，削除されていないクラスメンバを初期ソースプログラムに残す．このような方針では，マージ後のソースプログラムにおいて利用されないクラスメンバ（デッドコード）が発生することがある．しかしながら，テストにおける実行経路を調べることにより，このようなクラスメンバを後で削除することは容易である．

上記の方針を S_A と S_B に適用すると，S_{Ij}（$\in S_I$）は以下のようになる．

$$S_{Ij} = \langle\, m_{A1}, m_{B1}, m_{A2}|m_{B2}, m_{B3}, m_{A3}|m_{B4} \,\rangle$$

m_{A2} と m_{B2} は，それらのシグネチャが同じで両方が独立して書き換えられているクラスメンバである．また，m_{A3} と m_{B4} は，それらのシグネチャが同じで両方とも新規に追加されたクラスメンバである．その他のクラスメンバは，同一のシグネチャを持つクラスメンバが存在しないので，そのまま S_{Ij} に含まれる．排他的に選択されるクラスメンバを展開することで，以下に示す 4 つのクラスメンバの集まりが得られる．

$S_{I1} = \langle\, m_{A1}, m_{B1}, m_{A2}, m_{B3}, m_{A3} \,\rangle$
$S_{I2} = \langle\, m_{A1}, m_{B1}, m_{B2}, m_{B3}, m_{B4} \,\rangle$
$S_{I3} = \langle\, m_{A1}, m_{B1}, m_{A2}, m_{B3}, m_{B4} \,\rangle$
$S_{I4} = \langle\, m_{A1}, m_{B1}, m_{B2}, m_{B3}, m_{A3} \,\rangle$

4 つの初期ソースプログラム S_{I1}，S_{I2}，S_{I3}，S_{I4} は，それぞれのクラスメンバを並べて記述することで作成する．

このようにして作成されたそれぞれの初期ソースプログラムには，クラスメンバに対する参照エラーが含まれることがある．そこで，APR-MR は，それぞれの初期ソースプログラムを実際にコンパイルすることで，実行できないソースプログラムを APR の適用前に除外する．本例では，4 つの初期ソースプログラムがすべてコンパイル可能であるため，$S_I = \{\, S_{I1}, S_{I2}, S_{I3}, S_{I4} \,\}$ となる．

3.2 APR による初期ソースプログラムの修正

多くの場合，クラスメンバの組合せによって作成した初期ソースプログラム S_{Ij}（$\in S_I$）は，マージ後のソースプログラム S_M の振舞いを定義するテストスイート T_M を満たさない．このステップでは，S_{Ij} を APR における修正対象とし，T_M を満たすソースプログラム S_M を出力する．ここで，S_M は 1 個とは限らない．

APR システムとして，kGenProg [21]（1.5.5 版）[*2] を改造して利用した．改造点は 2 つである．1 つ目は，修正材料の指定に関する改造である．kGenProg では，修正材料を同一ファイル内，同一パッケージ内，同一プロジェクト内から選択できるようにオプションが用意されている．これに対して，APR-MR では，マージ対象のソースプログラム S_A と S_B を修正材料とする．そこで，我々は，kGenProg

[*2] https://github.com/kusumotolab/kGenProg

に対して，修正材料を指定するオプションを追加した．これにより，ファイル単位，クラス単位，メソッド単位で修正材料となるソースプログラムを指定できる．指定するソースプログラムは構文解析に成功すればよく，クラス名が重複しているような静的な意味エラーを含んでいてもよい．

2つ目は，条件式の置換に関する改造である．我々は，Liu らの考察 [22] や Ghiotto らの調査結果 [20] から，マージ競合の解決は Java プログラムの文レベルの挿入，削除，置換操作による変異だけでは難しいと考えた．実際，Wen らは，抽象構文木の要素タイプに基づき，文よりも粒度の細かい式に対する変異操作を行う APR システムを提案している [23]．残念ながら，APR-MR を考案した時点で採用した kGenProg（1.5.5 版）は文レベルの変異操作しか実現していなかった．そこで，kGenProg に対して，条件式の置換による変異操作を追加した．

具体的には，疑惑値（バグを含む可能性）に基づき変異対象として選択された文が条件式を含む（**if**，**while**，**do**，**for**，**switch** の）場合，文レベルでの挿入，削除，置換だけでなく，その文の条件部に現れる式レベルでの置換をランダムに実施する．一方，文の条件部に現れない式は置換対象としない．置換対象が式の場合，置換において再利用する式も修正材料に含まれる文の条件部から取得する．また，条件部に現れる式が 2 項演算子で結合されている場合，もとの式および結合されている 2 つの式を再帰的に収集し，置換対象あるいは再利用対象とする．このような拡張により，従来の 3 つの変異操作（文の挿入，文の削除，文の置換）に加えて，条件式の置換という 4 つ目の変異操作を実現している．

4. 実験

APR-MR がマージ競合を自動解決できる能力を持つことを確認するために，以下に示す 3 つの実験を行った．

実験 A 単一のソースファイルを変更した際に発生するマージ競合（図 1 の例題）を用いた実験

実験 B 複数のソースファイルを変更した際に発生するマージ競合を用いた実験

実験 C オープンソースプロジェクトにおけるマージ競合を用いた実験

実験 A，B，C では，マージ対象となるソースプログラムの規模（総行数）とソースプログラムの変更行数が異なる．実験 B は，実験 A に比べてマージ対象となるソースプログラムの規模が大きく，ソースプログラムの変更行数も多い．実験 C における競合行数は実験 A や実験 B に比べてそれほど多くないものの，プロジェクト全体に含まれるファイル数は多く，マージ対象のソースプログラムの規模はきわめて大きい．

すべての実験は，3.7 GH Quad-Core Intel Xeon E5 CPU とメモリ容量 64 GB を搭載した MacPro（macOS 10.15）で実施した．Java VM には JRE 11.0.6 を利用し，VM に 16 GB のメモリ容量を割り当てた．本実験では，初期ソースプログラムの作成は手動で機械的に（確実に自動化できる方法で）行った．

ここでは，競合が解決された（APR により修正に成功した）ソースプログラムをマージ後のソースプログラムと呼ぶ．実験 A，B，C の kGenProg の実行において，マージ後のソースプログラムをなるべく数多く出力するために，修正パッチの最大数（required-solutions オプションの値）を比較的大きな値である 100 と設定した．また，修正パッチの生成において，10 個の値（0〜9）を乱数発生のためのシードとして指定し，kGenProg を 10 回起動した際のデータを記録した．さらに，すべての実験において，欠陥限局（fault localization）手法には Ochiai [24] を用いた．実験データを https://www.fse.cs.ritsumei.ac.jp/APR-MR に示す．

4.1 実験 A

この実験では，3.1 節で説明した 4 つの初期ソースプログラム S_{I1}, S_{I2}, S_{I3}, S_{I4} を用意して，自動マージを実施した．**表 1** に，実験 A において出力されるソースプログラムの数と出力に費やす合計時間を示す．G は世代を打ち切る最大数（`max-generation` オプションの値）を指す．また，V は 1 世代あたりの変異の生成数（`mutation-generating-count` と `crossover-generating-count` オプションの値）を指す．$\#S_M$ は，乱数のシードを変えた 10 回の起動により出力されたマージ後のソースプログラムの総数を指す．異なるソースファイルに対して異なるパッチを適用することで，内容が同一のソースプログラムが出力されることがあるため，$\#S_M$ の数は重複を含む．括弧内の数値はマージ後のソースプログラムのうち，著者らが受け入れ可能と判断したものの数を指す．T は 10 回の起動において S_M を出力するために費やす合計時間である．初期ソースプログラムを作成する時間と受け入れ可否を判断する時間は含まれない．

一般的に，マージ競合の解決方法は唯一ではないが，実験をはじめる前に受け入れ可能な 2 つのソースプログラム（図 2 の $S_{M\text{-}A1}$ と $S_{M\text{-}A2}$）を用意しておいた．また，可読性の観点から，以下の条件を満たすソースプログラムを機械的に受け入れ不可とした．

(1) 参照されない変数への代入文を含む．
(2) 検査が不要な（必ず真になる）条件式を持つ **if** 文を含む．
(3) 状態を変化させないメソッドへの重複する呼び出し文を含む．

これら 3 つの条件を判定する手間を削減するため，我々は簡単な静的解析ツールを作成し，受け入れ不可なソースプログラムを目視の前に除外した．実験 A において，除外されずに残ったソースプログラムは，$V = 20, 50, 100$ に

表 1 実験 A において出力されるソースプログラムの数（$\#S_M$）と出力に費やす合計時間（T）
Table 1 Numbers of merged source programs ($\#S_M$) and execution times for producing them (T) in Experiment A.

変異の生成数 V	世代数 $G=10$		世代数 $G=20$		世代数 $G=30$		世代数 $G=40$		世代数 $G=50$	
	$\#S_M$	T（秒）	$\#S_M$	T（秒）	$\#S_M$	T（秒）	$\#S_M$	T（秒）	$\#S_M$	T（秒）
20	78 (2)	153	213 (2)	261	381 (3)	386	625 (3)	532	803 (3)	703
50	242 (5)	259	833 (5)	521	1,471 (5)	827	1,801 (5)	1,068	1,967 (5)	1,345
100	590 (5)	418	1,498 (5)	903	1,824 (5)	1,332	1,990 (5)	1,773	2,062 (5)	2,222

対して，それぞれ 12 個，14 個，16 個であった．これらのソースプログラムに対して，第 1 著者が目視で受け入れ可否に対する最終判断を行った．

表 1 に示すように，今回の実験では最大 5 個の受け入れ可能なソースプログラムが出力されている．5 個のうちの 3 個（ここでは，$S_{M\text{-}C1}$，$S_{M\text{-}C2}$，$S_{M\text{-}C3}$ と呼ぶ）を図 4 に示す．＋は追加された行，－は削除された行を指す．これらは，それぞれ S_{I1}，S_{I2}，S_{I4} を修正したものである．

本実験では，あらかじめ用意した $S_{M\text{-}A1}$ と同一のソースプログラム（図 4 の $S_{M\text{-}C1}$）が，世代の打ち切り数 G の値および変異の生成数 V の値に関係なく出力できた．$S_{M\text{-}A2}$ については，G の値や V の値を増やしても，類似するソースプログラム（たとえば，図 4 の $S_{M\text{-}C2}$）しか出力できなかった．また，あらかじめ用意していなかったが，受け入れ可能なソースプログラム（たとえば，図 4 の $S_{M\text{-}C3}$）が，V の値を増やすことで 2 個出力できた．

4.1.1 実験 A の総括

表 1 を見ると，G の値の増加は受け入れ可能なソースプログラムの増加にあまり貢献していない．これに対して，V の値を増やすことで（実験 A では $V=50$ で十分であるが），受け入れ可能なソースプラグラムの出力数が増加していることが分かる．

実行時間を見ると，5 個の受け入れ可能なソースプログラムを出力するのに，$G=10$，$V=50$ のとき最短で 259 秒（4 分 19 秒）費やしていた．このとき出力される 242 個のソースプログラムのうち，静的解析ツールによる除外（約 19 秒）後に残った数は 7 個であった．結果として，5 分以内にこれら 7 つのマージ後のソースプログラムを手に入れることができ，最終的に人手により 5 つが受け入れ可能と判断された．

ここで，マージ対象のソースプログラムを修正材料に利用せず，修正材料を初期ソースプログラムに限定した（改造前の kGenProg を利用した）場合，マージ後のソースプログラムは 1 つも出力されなかった．このことより，修正材料の選び方という観点で，APR-MR の改造が有効であることが確認できた．

$S_{M\text{-}C1}$

```
public class Book {
  private String priceInfo;
  private int price;
  public Book(int price) {
    if (price > 0) {
      priceInfo = String.valueOf(price) + "-yen";
+     this.price = price;
    } else {
      priceInfo = "Not on sale";
    }
  }
  public int getPrice() {
    return price;
  }
  public String getPriceInfo() {
    return priceInfo;
  }
}
```

$S_{M\text{-}C2}$

```
public class Book {
  private String priceInfo;
  private int price;
  public Book(int price) {
    if (price > 0) {
      this.price = price;
    }
  }
  public int getPrice() {
    return price;
  }
  public String getPriceInfo() {
-   return priceInfo;
+   if (price > 0) {
+     priceInfo = String.valueOf(price) + "-yen";
+   } else {
+     priceInfo = "Not on sale";
+   }
+   return priceInfo;
  }
}
```

$S_{M\text{-}C3}$

```
public class Book {
  private String priceInfo;
  private int price;
  public Book(int price) {
-   if (price > 0) {
-     this.price = price;
-   }
+   priceInfo = "Not on sale";
+   if (price > 0) {
+     priceInfo = String.valueOf(price) + "-yen";
+     this.price = price;
+   }
  }
  public int getPrice() {
    return price;
  }
  public String getPriceInfo() {
    return priceInfo;
  }
}
```

図 4 APR-MR によって出力されたマージ後のソースプログラム
Fig. 4 Source programs merged by APR-MR.

4.2 実験 B

この実験では，複数のファイルをマージする際の競合

表 2 実験 B におけるマージ競合と競合に関係するソースプログラム S_O，S_A，S_B の情報
Table 2 Information on occurring merge conflicts in Experiment B and source programs S_O, S_A, and S_B related to the conflicts.

例題	競合するソースファイル	競合するメソッド	競合行数	S_O の行数	S_A の変更行数	S_B の変更行数
VS1	`Customer.java`	`statement()`	7	425	+313−35	+57−10
VS2	`Customer.java`	`statement()`	7	425	+315−44	+45−14
DT	`DrawCanvas.java`	5 個のメソッド†	37 (= 27 + 10)	2,610	+479−24	+168−6

† `mousePressed()`, `grabFigure()`, `drawFigure()`, `createNewFigure()`, `showPopup()`

表 3 実験 B において出力されるソースプログラムの数（$\#S_M$）と出力に費やす合計時間（T）
Table 3 Numbers of merged source programs ($\#S_M$) and execution times for producing them (T) in Experiment B.

例題	変異の生成数 V	世代数 $G = 10$		世代数 $G = 20$		世代数 $G = 30$		世代数 $G = 40$		世代数 $G = 50$	
		$\#S_M$	T（秒）	$\#S_M$	T（秒）	$\#S_M$	T（秒）	$\#S_M$	T（秒）	$\#S_M$	T（秒）
VS1	20	2 (0)	85	3 (0)	140	3 (0)	193	3 (0)	267	4 (0)	345
	50	2 (0)	87	3 (0)	156	3 (0)	229	3 (0)	288	4 (0)	444
	100	45 (1)	295	74 (1)	540	117 (1)	847	138 (1)	1,258	179 (1)	1,757
VS2	20	1 (1)	128	2 (1)	193	2 (2)	239	5 (3)	297	6 (3)	354
	50	3 (2)	201	6 (2)	323	6 (2)	454	6 (2)	579	8 (3)	717
	100	6 (3)	300	12 (5)	533	14 (5)	796	15 (5)	1,102	18 (5)	1,411
DT	20	3 (1)	455	7 (1)	888	12 (2)	1,364	15 (2)	1,890	16 (2)	2,433
	50	11 (2)	858	26 (2)	1,961	33 (2)	3,304	40 (2)	4,667	69 (2)	5,801
	100	15 (2)	1,463	30 (2)	3,602	38 (2)	5,856	38 (2)	6,233	38 (2)	6,229

の解決を試みる．この実験で利用する 3 つの例題（VS1，VS2，DT）におけるマージ競合の情報を**表 2** に示す．VS1 と VS2 では，Fowler の書籍 [25] に掲載されている Video Store のリファクタリング例を利用した．DT は大学の演習課題で利用する簡単な図形描画ツールである．表 2 において，S_O は変更前のソースプログラム，S_A と S_B はそれぞれ S_O を編集したマージ対象のソースプログラムである．競合するソースファイルと競合するメソッドは Git により報告された競合に関係するファイルとメソッドの名前を指す．競合行数は，競合報告において “`<<<<<<<`” と “`=======`” に挟まれる行数と “`=======`” と “`>>>>>>>`” に挟まれる行数を足した数である．DT では，複数のメソッドにまたがる 2 カ所で競合が報告された．S_O の行数はテストファイルの行数も含む．S_A と S_B の変更行数は，S_O に対する追加行数（+）と削除行数（−）を指す．

表 3 に，実験 B において出力されるソースプログラムの数と出力に費やす合計時間を示す．G，V，$\#S_M$，T の意味は実験 A と同一である．以下，VS1，VS2，DT について結果を述べる．

4.2.1 VS1 における結果

VS1 において，S_A における変更はリファクタリング，S_B における変更は機能改変である．VS1 では，初期ソースプログラムを 2 つ作成したが，そのうちの 1 つはそのままでマージ後のソースプログラムに対するテストケースを通過した．これは，S_A における変更がリファクタリングであり，用意したテストケースにおいて外部的振舞いを保存したためである．

リファクタリングがテスト結果に影響を与えない場合，マージにおいてリファクタリングによるコード変更を取り入れなくても，テストは成功する．このような初期ソースプログラムに対して，kGenProg はパッチの生成を行わないため，APR-MR では，これをそのまま S_M として出力する．その一方で，ソースプログラムの改善という観点からは，リファクタリングを単純に無視するマージ競合の解決は好まれないことが多い．VS1 では，このような状況において，もう片方の初期ソースプログラムを利用することで，リファクタリングによるコード変更を取り入れたマージ競合の解決ができるかどうかを確認している．

表 3 を見ると，$V = 100$ のとき，マージ後のソースプログラムが数多く出力されている．そこで，実験 B でも，実験 A のときと同様に静的解析ツールによる選別を行ったところ，ソースプログラムは 1 つしか残らなかった．このソースプログラムの内容を確認したところ，簡潔な変更（4 行の追加と 1 行の削除）で競合を解決しており，受け入れ可能であることが確認できた．

本実験では，受け入れ不可なすべてのソースプログラムが機械的に除外された．競合行数は 7 行と短く，人間による競合の解決は容易であるものの，APR-MR を利用することで，開発者は出力されるマージ後のソースプログラムを確認するだけでマージ競合が解決できる．

4.2.2 VS2 における結果

VS2 では，VS1 と同様に，S_A における変更がリファク

タリングである．しかしながら，S_B における機能改変のための変更が S_A のリファクタリングによる外部的振舞いの保存を破壊する．このため，作成した 2 つの初期ソースプログラムは両方ともテストに失敗する．

表 3 に示すように，VS2 におけるマージ後のソースプログラムの数はそれほど多くなかった．そこで，すべてのソースプログラムを第 1 著者が目視で確認して，受け入れの可否を判断した．最も簡潔な変更（1 行の追加と 1 行の削除）で達成されたソースプログラムが，$G = 10$，$V = 50$ のときに出力されている．この変更は，Git による 3-way マージにおいて競合として報告される 7 行，さらにそれら 7 行を含むメソッドにも含まれない．このような競合を解決するためには，人間がテスト結果からマージにおいて変更が必要なソースファイルを特定する必要がある．

本実験では，VS2 に含まれるソースファイルの数が 7 個（テストファイルを除く）と少なかったため，変更箇所の特定は困難であったとは考えにくい．しかしながら，競合に関係するソースファイルの数が多い場合，変更箇所の特定は容易とはいえない．特に，本実験のように，変更箇所が競合報告に含まれない場合，たとえマージ競合の解決における変更の規模が小さくても，開発者の負担は大きくなる可能性がある．このような場面において，人手の介入を抑える APR-MR の活用は有効である．

4.2.3 DT における結果

DT は，図形描画ツール利用時のマウスボタンに対する同一のイベントに対して，異なる機能（図形の移動と消去）が独立して拡張される例題である．マージ競合の解決では，2 つの機能を切り替える **if** 文を矛盾なく混ぜ合わせる必要がある．本実験では，マージ競合の自動解決によって，この作業が自動化できるかどうかを確認している．

表 2 に示すように，DT では，同一ファイルに存在する 5 個のメソッドがマージ競合に関係し，競合行数も合計 37 行と多い．また，マージ対象のソースプログラムの行数はどちらも 2,000 行を超えている．このような状況において，実験 B の VS1 および VS2 と同様に，競合を解決するために必要な修正箇所の範囲について限定せず，また競合するソースファイル全体を修正材料に指定して，APR-MR の適用を試みた．結果として，このような設定では修正のための探索空間が膨大になり，出力されたソースプログラムの数は少なかった（$V = 20$ のとき 0 個，$V = 50$ のとき最大で 1 個，$V = 100$ のとき最大で 4 個）．これらを第 1 著者が目視で確認したところ，複数のメソッドが不必要に修正されており，受け入れ可能とはいえなかった．

そこで，修正箇所の範囲と修正材料を，テストに失敗するメソッド `mousePressed()` に限定し，マージ後のソースプログラムの出力を再度試みた．その結果，V の値の大小にかかわらず，G の値が小さくても（早い世代で）受け入れ可能なソースプログラムの出力に成功した．この例題は，GUI アプリケーションであり，テスト実行のたびに GUI ウインドウの作成が行われ，それに多大な時間を費やしていた．また，マージ競合の解決において，GUI ライブラリの提供する API への呼び出しが数多く追加されていた．残念ながら，追加された API 呼び出しが不要であるかどうかを単純な静的解析で判定することは難しく，実験 A で利用した静的解析ツールでは受け入れ不可なソースプログラムを除外することはできなかった．このため，受け入れ可否の判断は出力されたすべてのソースプログラムに対して第 1 著者が目視で実施した．本実験を通して，マージ競合の解決において，やや複雑な制御構造（**if** 文）を混ぜ合わせる変更が自動的に実施できることを示すことに成功した．

4.2.4 実験 B の総括

実験 B における 3 つの例題を通して，APR-MR はマージ競合を自動的に解決できる可能性があることが分かる．実験 B においても，実験 A のときと同様に，マージ対象のソースプログラムを修正材料に利用しない場合に自動マージは成功しなかった．このことは，修正材料の選び方を改良したことの有効性を示している．

さらに，APR-MR では，kGenProg を改造することで，文だけでなく条件式の置換による変異操作を可能としている．この効果を確認するために，文だけを変異操作の対象とするように制限した kGenProg を別途用意し，3 つの例題に対して，マージ後のソースプログラムの出力を試みた．その結果，VS1 と DT については，マージ後のソースプログラムの出力に成功した．これに対して，VS2 では，マージ後のソースプログラムが 1 つも出力されなかった．出力されたソースプロラムの数や質，および実行時間に関する考察は未着手であるが，条件式の置換による変異操作を導入したことの効果を示す例題が得られたことは，有用性の観点から意義がある．

4.3 実験 C

実験 C では，実際の Java プロジェクトにおいて，APR-MR に実用性があるのかどうかを確認する．そこで，我々は，Defects4J [*3] のプロジェクトを対象に選び，マージ競合の解決を試みた．Defects4J は，APR 技法の性能を比較する際のベンチマークとして頻繁に利用されている．

まず，15 個のプロジェクト[*4] の 2020 年 3 月 12 日時点の履歴に対して，過去のマージ（親を 2 つ持つコミット）を再演し，マージ競合を検出した．次に，マージ競合が発生していた 12 個のプロジェクトに対して，競合が Java ソースコードで発生していること，および競合する Java ソースファイルに対して末尾に “`Test`” が付与されたファイル

[*3] https://github.com/rjust/defects4j

[*4] JacksonDatabind プロジェクトと JacksonXml プロジェクトは規模が大きく，マージ競合の分析に長い時間がかかるため対象から除外した．

表 4 実験 C におけるマージ競合と競合に関係するソースファイル F_O，F_A，F_B の情報
（F_O，F_A，F_B はそれぞれ S_O，S_A，S_B に存在するソースファイル）

Table 4 Information on occurring merge conflicts in Experiment C and source files F_O, F_A, and F_B related to the conflicts
(F_O, F_A, and F_B are included within S_O, S_A, and S_B, respectively).

競合	競合するソースファイル	競合するメソッド	競合行数	F_O の行数	F_A の変更行数	F_B の変更行数
Jsoup-1	`QueryParser.java`	`byTag()`	3	356	+12−6	+2−2
Jsoup-7	`HttpConnection.java`	`createConnection()`	11	807	+90−3	+25−23

名を持つテストファイルが存在するかどうかを調べた．その結果，8 個のプロジェクトにおいて，これらの条件を満たす合計 41 個のマージ競合を見つけた．

これらの競合に対して，1 つずつ内容を確認したところ，マージ対象のソースプログラムの一方あるいは両方がコンパイルエラーであった競合，あるいはマージコミットが受け入れられた（正解と見なす）ソースプログラムがもともとテストケースを満たさなかった競合が 17 個であった．他にも，コメントの競合が 3 個，メソッド内部で発生していない競合が 15 個であった．残念ながら，これらの競合については APR-MR の対象外であり，その解決は見込めない．

残りの 6 個のマージ競合において，マージ後のソースプログラムに対するテストケースをそのまま実行したところ，マージ対象のソースプログラムのどちらに対しても失敗した．よって，これらは振舞い競合である．内訳は，Jsoup プロジェクトが 2 個，Lang プロジェクトが 1 個，Mockito プロジェクトが 3 個である．これらの 6 個のマージ競合に対して，APR-MR の適用を試みたところ，Lang プロジェクトと Mockito プロジェクトについて kGenProg の実行に失敗した[*5]．結果として，Jsoup プロジェクトの 2 個のマージ競合において，APR-MR によりマージ後のソースプログラムが出力できた．その際，実験 B の DT のときと同様の理由により，修正コード範囲と修正材料をテストに失敗したメソッドに限定した．

表 4 に，実験 C で利用する Jsoup プロジェクトにおける 2 つのマージ競合の情報を示す．Jsoup-1 および Jsoup-7 は，このプロジェクトにおける全部で 7 個のマージ競合を時間順に並べたときの 1 番目と 7 番目のマージ競合を指す．競合するソースファイル，競合するメソッド，競合行数の意味は，表 2 と同じである．ここで，F_O は，プロジェクト内のソースプログラム全体ではなく，S_O に存在する変更前ソースファイルを指す．また，F_A，F_B は，それぞれ S_A，S_B において競合する変更後のソースファイルを指す．F_A と F_B の変更行数は，F_O に対する追加行数（+）と削除行数（−）を指す．

本実験では，実験 A において $G = 30$ 以降で受け入れ可能なソースプログラムの数が増加しないという結果をふまえ，打ち切りの世代数は 30 とした．**表 5** に，実験 C において出力されるソースプログラムの数，出力に費やす合計時間，正解判定の結果を示す．V，$\#S_M$，T の意味は実験 A と同一である．$\#S_U$ は，重複を排除したソースプログラムの数を指す．

[*5] 対象としている Java の版が異なる，一部のライブラリが不足しているなどが考えられるが，正確な原因は不明である．

表 5 実験 C において出力されるソースプログラムの数（$\#S_M$ と $\#S_U$），出力に費やす合計時間（T），正解に対する判定の結果

Table 5 Numbers of merged source programs ($\#S_M$ and $\#S_U$), execution times for producing them (T), and assessment for them in Experiment C.

競合	変異の生成数 V	$\#S_M$	$\#S_U$	T（秒）	判定結果
Jsoup 1	20	21	16	1,021	equiv
	50	44	38	2,646	as-is
	100	59	53	4,887	as-is
Jsoup 7	20	81	81	2,388	
	50	207	204	5,912	
	100	353	346	6,240	equiv

4.3.1 実験 C の総括

本実験では，プロジェクトにおいて受け入れられた（正解と見なす）マージ後のソースプログラムがすでに存在する．このため，出力されたそれぞれのソースプログラムに対して著者らが受け入れ可否を判断せず，正解のソースプログラムと内容を比較することで，APR-MR が受け入れ可能なソースプログラムを出力したかどうかを確認した．ここで，Jsoup-1 では，F_A に対して 2 行の追加と 2 行の削除で競合の解決が実現できる．F_B を修正した場合，12 行の追加と 6 行の削除が必要である．また，Jsoup-7 では，F_A に対して 25 行の追加と 23 行の削除，F_B に対して 90 行の追加と 3 行の削除で競合の解決が実現できる．

表 5 において，as-is は，正解のソースプログラムの内容と完全に一致するソースプログラムが出力されたことを指す．また，equiv は，文の順序が一部異なるものの，正解のソースプログラムと振舞いが同一のソースプログラムが出力されたことを指す．空欄は，as-is および equiv に該当するソースプログラムが出力されなかったことを指す．

表 5 を見ると，Jsoup-1 および Jsoup-7 のどちらに対しても，$V = 100$ とすることで，as-is あるいは equiv となるソースプログラムが出力されている．このことは，実際の

オープンソースプロジェクトにおいて，APR-MR がマージ競合を自動解決できる場面が存在することを示している．

その一方で，表 5 に示すように，ソースプログラムの出力には多大な時間を費やしている．Jsoup-1 では，F_A に対して最小で 4 行の修正で競合が解決するにもかかわらず，equiv の出力までに 1,021 秒（約 17 分），as-is の出力までに 2,646 秒（約 44 分）費やしている．Jsoup-7 では，F_A および F_B のどちらに対しても数多くの行の修正が必要（それぞれ 93 行と 48 行）であり，equiv の出力までに 6,240 秒（104 分）を費やしている．

4.4 実験全体に対する考察

実験全体を通して，マージにおける競合行数やマージ対象のソースプログラムの規模にかかわらず，著者らが受け入れ可能と判断したソースプログラムがマージ結果として出力されることが確認できた．特に，実験 C の結果は，実際のオープンソースソフトウェアプロジェクトにおいて，APR-MR が振舞い競合を自動的に解決できる可能性があることを示している．

遺伝的アルゴリズムに基づく APR を利用するという観点からは，1 世代あたりの変異の生成数を増やす方が，マージ競合の解決に有効であるという結果が得られた．その一方で，世代数を打ち切る最大数を増やしてもマージ結果として受け入れ可能なソースプログラムが増えないという知見も得られた．

さらに，マージ競合の解決においてソースプログラムの修正が少ない場合（たとえば，実験 A や実験 B の VS1）に，受け入れ不可なソースプログラムが大量に出力されるという点が明らかになった．そこで，本実験では，受け入れ不可なソースプログラムを機械的に除外する 3 つの条件を設定した．当初，これらの条件を検査する静的解析モジュールを APR-MR に組み込むことを考えた．しかしながら，解決時間を短縮するために，出力されたソースプログラムの一部を最後に除外する方がよいのか，それとも除外条件を遺伝的アルゴリズムにおける変異の生成アルゴリズムに直接組み込む方がよいのかに関する検討が不十分であるという理由から，現時点での組み込みは行わなかった．また，本実験で設定した 3 つの条件は，著者らの経験的に基づく単純なものである．より高い精度で受け入れ不可なソースプログラムを除外するためには，APR システムにより出力されるソースプログラムの正しさを自動査定する手法 [26] の導入を検討する必要がある．

実験 B の DT の結果や実験 C の結果を見ると，マージにおける競合行数が多い場合やマージ対象のソースプログラムの規模が大きい場合，競合を解決するために必要な修正箇所の範囲と修正材料を限定しなければならないことも判明した．これは，修正のための探索空間が膨大になることを避けるために必須である．しかしながら，修正箇所の範囲と修正材料の限定は容易ではない．たとえば，修正箇所を Git により報告される競合箇所に単純に限定してしまうと，実験 B の VS2 において受入れ可能なソースファイルが出力されない．実験 B の DT や実験 C のときに人手で行ったのと同様に，テスト結果に応じて修正箇所を自動的に限定する仕組みの確立が必要である．

以上より，プロジェクトの規模が小さい，あるいは，競合解決においてコードを修正する部分が局所的かつあらかじめ限定できる場合には，APR-MR の適用可能性は高いようにみえる．ただし，マージ競合の解決に長い計算時間を費やす場面があること，さらに出力されたソースプログラムを人間が判定する手間が必要なことを考えると，APR-MR が有用な場面はまだまだ少ないといえる．実験 C の結果を見ても，現時点では，開発者が統合開発環境上でマージを試みた時点から，マージ競合の解決をリアルタイムに支援するというような使い方には適さないことは明確である．APR-MR の有用性を高めるためには，マージ競合の解決における効率性を改善する必要があり，探索空間の削減や無駄な変異を生成しない仕組みの検討が必須である．また，APR-MR による競合の解決を夜間のバッチ処理で実施したり，実際にマージを行う前にバックグラウンドで自動的に APR-MR を起動しておいたりするなど運用上の工夫も求められる．

当然ながら，APR-MR によりマージ競合の自動解決ができるかどうかは，APR システムの自動修正能力に大きく依存している．よって，APR-MR の適用可能性や有効性を議論するためには，利用する APR システムの適用可能性や有効性を明らかにする必要がある．たとえば，Liu らは，Defects4J で提供されている 395 個のバグのうち，少なくとも 1 つ以上の APR システムで修正できたバグは全部で 101 個であったと報告している [27]．また，Durieux らは，Defects4J で提供されている 2,141 個のバグのうち，既存の 11 個の APR システムで修正できたバグの数は 15〜213 個であったと報告している [28]．これらの報告を見る限り，現時点における APR による修正能力には改善の余地が大きい．さらに，どのような特性を持つソースプログラムやプロジェクトに対して，APR システムが自動修正可能であるかについての調査も十分とはいえない．残念ながら，このような状況において，APR-MR が適用可能な範囲や有効である場面を明確に示すことは難しい．APR 技法の発展を待つのと同時に，APR 技法の応用という観点から改善点を明らかにしていくことが重要である．

4.5 妥当性の脅威

実験 A および B において，マージ後のソースプログラムのテストケースは著者らが用意した．APR システムは与えられたテストケースの実行に成功するようにソースプログラムを修正する．よって，修正後のソースプログラ

ムが必ずしも開発者の望む振舞いを満たしている（与えられていないテストケースの実行に成功する）とは限らない [29], [30], [31], [32], [33]．用意するテストケースによって，実験結果が変わる可能性は高い．さらに，実験 A および B において，出力されたソースプログラムが受け入れ可能かどうかの判定は第 1 著者の主観に基づいている．判定者によっては，実験結果が変わる恐れがある．客観的な評価のためには，テストケースを機械的に生成することで，出力されたソースプログラムを評価する手法 [34], [35], [36] の利用が考えられる．

欠陥限局手法によって，APR システムの修正能力が変わることが報告されている [37], [38], [39]．よって，実験で用いた Ochiai とは異なる欠陥限局手法を利用することで，実験結果が変わる可能性がある．また，実験 B の DT および実験 C における修正箇所の範囲の指定および修正材料の探索範囲の指定は，著者らの経験に基づいて実施した．それぞれの範囲を指定したことが，欠陥限局および修正能力の向上に有利に作用した可能性がある．

実験 B では，汎用的な例題の採用を意識した．しかしながら，著者らが例題と提案手法の両方を設計しているため，例題が APR-MR に有利になっている可能性が否定できない．また，現実的な時間で自動マージができることを確認するために，実験 A と実験 B における例題の規模は小さい．出力されるマージ後のソースプログラムの質や数，および実行時間の観点による評価を一般化するためには，実際のプロジェクトで発生するマージ競合を利用したさらなる実験が必要である．このために，文献 [20], [40], [41], [42], [43], [44] に報告されている競合を調査し，振舞い競合を収集することを考えている．さらに，文献 [45] と同様に，ミューテーションテストツールを利用することで，競合を機械的に挿入したソースプログラムを用意することも考えられる．

5. 関連研究

ここでは，マージを支援する手法やツールを紹介する．さらに，APR-MR で採用した，あるいは採用できる可能性のある APR システムについて取り上げる．

5.1 マージ支援手法

行単位のテキスト差分に基づく 3-way マージを実現したツールに，`git-merge` や `kdiff3` がある．これらのツールは，ソースプログラムの構造を考慮しないため，プログラミング言語を選ばないという利点を持つ．その一方で，このようなテキスト的なマージは，構文誤りを含むソースプログラムを生成してしまう．

これに対して，ソースプログラムの構造を考慮したマージ手法が提案されている．Westfechtel の手法は，ソースプログラムの構文要素とそれらの関係に対するマージ規則を設定することで，構文的に正しいマージ後のソースプログラムを生成する [46]．Buffenbarger の手法は，変数や関数の宣言と利用関係を保存するように，マージ後のソースプログラムを生成する [47]．Nui らの手法は，ソースプログラムにおける関数とその呼び出しをグラフで表現し，圏論（category theory）における射（morphism）に基づき関数の呼び出し関係を決定することで，マージを実現する [48]．また，Apel らは，マージを試みる開発者が，構造を考慮しない（非構造的）マージを行うのか，あるいは構造を考慮した（構造的）マージを行うのかを自由に選択できる半構造的マージを提案した [41], [49], [50]．文献 [51] では，構造的マージに対する半構造的マージの優位性が示されている．Shen らの手法は，ソースプログラムに含まれるプログラム要素（構文要素のインスタンス）間の関係をグラフで表現し，節点や矢印の追加と削除に基づきグラフを合成することで，ソースプログラムをマージする [44]．Lippe らの手法は，ソースプログラムに対する変更をコード変換（編集操作）で表現し，変換の列を混ぜ合わせることでマージを実現する [52]．この手法を実現したツールとして，MolhadoRef [53] や MergeHelper [54] が提案されている．これらの手法やツールはどれも構文的競合や静的な意味的競合の解決を目指しており，振舞い競合の解決を対象としていない．

振舞い競合の検出を目的とした手法やツールも提案されている．Horwitz らは，プログラム依存グラフを用いることで，ソースプログラム中に現れる変数の計算過程（スライス）に対する干渉（interference）を定義した [55]．独立に変更した 2 つのソースプログラムから生成したスライスが干渉する場合，振舞い競合が発生していると判定する．Sarma らの手法では，クラス（とそのメンバ）のシグニチャが変更された場合，そのクラスとそれを利用しているコードが競合していると見なす [56]．Pastore らの手法は，ソースプログラムを実行することで獲得した変数の値に関する制約（不変条件）[57] の変化を検査することで，振舞い競合を検出する [45]．Marcelo らは，ソースプログラムの差分に現れる変数の入出力関係（事前・事後条件）に基づき，3-way マージにおいて振舞い競合が発生しない条件を定義し，それを検査するツール SafeMerge を提案した [40]．Nguyen らの手法は，独立に変更された複数のソースプログラムの差分（コード断片）に対応する条件文を用意し，それぞれの差分を条件文のブロックとして混ぜ合わせた単一のソースプログラムに変換する [58]．変換後のソースプログラムに対してテストを実施することで，失敗する条件式の組合せが見つかった場合，振舞い競合が発生していると判定する．また，Silva らは，振舞い競合を正しく検出するためにはあらかじめ用意されたテストケースだけでは不十分であることを立証するために，Randoop [59] や EvoSuite [60] などのツールにより自動生成したテストケー

スにより，より多くの振舞い競合が検出できることを示している [13]．これらの手法やツールは，振舞い競合の検出や振舞い競合が発生しない場合の自動マージを実現しているものの，APR-MR が対象としている振舞い競合の自動解決は実現していない．

5.1.1 APR システム

APR-MR で採用した kGenProg は，遺伝的プログラミングに基づく GenProg [16], [61] および jGenProg [62] の後継となる APR システムである．kGenProg は，メモリ上でのコード変換（変異個体の生成）とテスト実行（変異個体の検証）の並列化を実現することで，高速なバグ修正を達成している [63]．さらに，将来的な改変を意識して設計されているため，拡張が容易となっている．APR-MR では，拡張容易性を重視して kGenProg を改造した．

GenProg 以外の生成&検証に基づく探索ベースの APR システムがいくつも提案されている．たとえば，プログラムの等価性に基づき探索空間を削減した AE [64]，ランダム探索を利用した RSRepair [65]，探索に過去のバグの履歴を利用する HistoricalFix [66]，ソースプログラムから学習した確率モデルに基づく探索を利用する Prophet [31] がある．他にも，人間による修正に基づき作成したテンプレートを利用する PAR [67]，コード変換のためのスキーマを利用した SPR [68]，修正対象を条件式に限定した ACS [69]，メソッド呼び出しの積極的な置換と機械学習を利用した Elixir [70]，テストにおける実行時情報に基づき修正ソースプログラムの候補の生成を抑える SketchFix [71]，複数の観点を取り入れた遺伝的プログラミングに基づく ARJA [72] がある．

生成&検証に基づく手法とは異なり，プログラムの意味論に基づく手法も提案されている．この手法は，テストケースから修正後のソースプログラムが満たすべき要求（制約）を抽出し，要求を満たすようにソースプログラムの一部を合成する．このため，修正後のソースプログラムの候補を大量に生成する必要がなく，ソースプログラムの修正が効率的に実施できる．たとえば，この手法を実装した Java のソースプログラムを対象とした APR システムに，JFix [73] や Genesis [74] がある．

6. おわりに

マージ競合の解決は，開発者にとって面倒な作業であり，その作業の自動化への期待は大きい．本論文では，自動プログラム修正を活用した振舞い競合の解決手法 APR-MR を提案した．3 つの実験において，受け入れ可能なマージ後のソースプログラムの出力に成功した．

実際のソフトウェア開発に APR-MR を導入するためには，受け入れ不可なソースプログラムを自動的に除外する仕組みの導入と実行時間の短縮が必須である．ここで，APR-MR は，生成&検証戦略の APR システムの利用を前提としていない．出力されるマージ後のソースプログラムの質の向上および数の抑制，出力までに費やす時間の短縮という観点で，さまざまな APR システムの活用を検討している．

3 章で説明した仕組みを見ると分かるように，マージ後のソースプログラムに対するテストケースさえ存在すれば，原理的には APR-MR は構文的競合や静的な意味的競合の解決を対象とすることも可能である．開発者から見ると，どのような競合が発生しているのかを事前に見極めることは面倒であり，APR-MR が対象とする競合の種類を増やすことの利益は大きい．その一方で，ソースプログラムに対する静的解析で解決できる可能性が高い構文的競合や静的な意味的競合を解決するために，テストを実施する戦略は非効率にみえる．今後，構文的競合や静的な意味的競合を含むソースプログラムを用いた実験を行い，マージ後のソースプログラムをどの程度の実行時間で出力可能であるか，さらに出力されたソースプログラムを開発者が受け入れ可能かどうかに関する調査を行う予定である．そのうえで，発生している競合に合わせて，構文的競合や静的な意味的競合を解決する既存ツールとの使い分けを自動的に行う仕組みを探求する．さらに，初期ソースプログラムの作成にそれらのツールを活用することも検討する．

謝辞 本研究の一部は，科研費（20H04166）の助成を受けたものである．kGenProg の利用に関する助言をいただいた肥後芳樹氏と柗本真佑氏に感謝する．

参考文献

[1] O'Sullivan, B.: Making Sense of Revision-Control Systems, *Comm. ACM*, Vol.52, No.9, pp.56–62 (2009).
[2] Muşlu, K., Bird, C., Nagappan, N. and Czerwonka, J.: Transition from Centralized to Decentralized Version Control Systems: A Case Study on Reasons, Barriers, and Outcomes, *Proc. ICSE '14*, pp.334–344 (2014).
[3] Bird, C. and Zimmermann, T.: Assessing the Value of Branches with What-If Analysis, *Proc. FSE '12*, pp.45:1–11 (2012).
[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] Perry, D.E., Siy, H.P. and Votta, L.G.: Parallel Changes in Large-scale Software Development: An Observational Case Study, *ACM Trans. Softw. Eng. Methodol.*, Vol.10, No.3, pp.308–337 (2001).
[6] Phillips, S., Sillito, J. and Walker, R.: Branching and Merging: An Investigation into Current Version Control Practices, *Proc. CHASE '11*, pp.9–15 (2011).
[7] Sarma, A., Redmiles, D. and van der Hoek, A.: Palantír: Early Detection of Development Conflicts Arising from Parallel Code Changes, *IEEE Trans. Softw. Eng.*, Vol.38, No.4, pp.889–908 (2012).
[8] Kasi, B.K. and Sarma, A.: Cassandra: Proactive Conflict Minimization through Optimized Task Scheduling, *Proc. ICSE '13*, pp.732–741 (2013).
[9] Guimarães, M.L. and Silva, A.R.: Improving Early Detection of Software Merge Conflicts, *Proc. ICSE '12*,

pp.342–352 (2012).
[10] Brun, Y., Holmes, R., Ernst, M.D. and Notkin, D.: Early Detection of Collaboration Conflicts and Risks, *IEEE Trans. Softw. Eng.*, Vol.39, No.10, pp.1358–1375 (2013).
[11] Nelson, N., Brindescu, C., McKee, S., Sarma, A. and Dig, D.: The life-cycle of merge conflicts: Processes, barriers, and strategies, *Empirical Software Engineering*, Vol.24, No.5, pp.2863–2906 (2019).
[12] Brindescu, C., Ramirez, Y., Sarma, A. and Jensen, C.: Lifting the Curtain on Merge Conflict Resolution: A Sensemaking Perspective, *Proc. ICSME '20*, pp.534–545 (2020).
[13] Silva, L.D., Borba, P., Mahmood, W., Berger, T. and Moisakis, J.: Detecting Semantic Conflicts via Automated Behavior Change Detection, *Proc. ICSME '20*, pp.174–184 (2020).
[14] Gazzola, L., Micucci, D. and Mariani, L.: Automatic Software Repair: A Survey, *IEEE Trans. Softw. Eng.*, Vol.45, No.1, pp.34–67 (2019).
[15] Xing, X. and Maruyama, K.: Automatic Software Merging using Automated Program Repair, *Proc. IBF '19*, pp.11–16 (2019).
[16] Weimer, W., Nguyen, T., Goues, C.L. and Forrest, S.: Automatically Finding Patches Using Genetic Programming, *Proc. ICSE '09*, pp.364–374 (2009).
[17] Martinez, M., Durieux, T., Sommerard, R., Xuan, J. and Monperrus, M.: Automatic Repair of Real Bugs in Java: A Large-scale Experiment on the Defects4J Dataset, *Empirical Software Engineering*, Vol.22, No.4, pp.1936–1964 (2017).
[18] Koegel, M., Naughton, H., Helming, J. and Herrmannsdoerfer, M.: Collaborative Model Merging, *Proc. OOPSLA '10*, pp.27–34 (2010).
[19] Costa, C., Figueirêdo, J., Murta, L. and Sarma, A.: TIPMerge: Recommending Experts for Integrating Changes across Branches, *Proc. FSE '16*, pp.523–534 (2016).
[20] Ghiotto, G., Murta, L., Oliveira, M. and van Der Hoek, A.: On the Nature of Merge Conflicts: A Study of 2,731 Open Source Java Projects Hosted by GitHub, *IEEE Trans. Softw. Eng.*, Vol.46, No.8, pp.892–915 (2020).
[21] Higo, Y., Matsumoto, S., Arima, R., Tanikado, A., Naitou, K., Matsumoto, J., Tomida, Y. and Kusumoto, S.: kGenProg: A High-Performance, High-Extensibility and High-Portability APR System, *Proc. APSEC '18*, pp.697–698 (2018).
[22] Liu, K., Kim, D., Koyuncu, A., Li, L., Bissyandé, T.F. and Traon, Y.L.: A Closer Look at Real-World Patches, *Proc. ICSME*, pp.275–286 (2018).
[23] Wen, M., Chen, J., Wu, R., Hao, D. and Cheung, S.-C.: Context-aware Patch Generation for Better Automated Program Repair, *Proc. ICSE '18*, pp.1–11 (2018).
[24] Abreu, R., Zoeteweij, P. and van Gemund, A.J.C.: An Evaluation of Similarity Coefficients for Software Fault Localization, *Proc. PRDC '06*, pp.39–46 (2006).
[25] Fowler, M.: *Refactoring: Improving the Design of Existing Code*, Addison-Wesley (1999).
[26] Ye, H., Martinez, M. and Monperrus, M.: Automated Patch Assessment for Program Repair at Scale, *EMSE*, Vol.26, No.2 (2021).
[27] Liu, K., Koyuncu, A., Kim, D. and Bissyandé, T.F.: TBar: Revisiting Template-Based Automated Program Repair, *Proc. ISSTA '19*, pp.31–42 (2019).
[28] Durieux, T., Madeiral, F., Martinez, M. and Abreu, R.: Empirical Review of Java Program Repair Tools: A Large-Scale Experiment on 2,141 Bugs and 23,551 Repair Attempts, *Proc. ESEC/FSE '19*, pp.302–313 (2019).
[29] Qi, Z., Long, F., Achour, S. and Rinard, M.: An Analysis of Patch Plausibility and Correctness for Generate-and-Validate Patch Generation Systems, *Proc. ISSTA '15*, pp.24–36 (2015).
[30] Smith, E.K., Barr, E.T., Goues, C.L. and Brun, Y.: Is the Cure Worse Than the Disease? Overfitting in Automated Program Repair, *Proc. ESEC/FSE '15*, pp.532–543 (2015).
[31] Long, F. and Rinard, M.: An Analysis of the Search Spaces for Generate and Validate Patch Generation Systems, *Proc. ICSE '16*, pp.702–713 (2016).
[32] Le, X.B., Thung, F., Lo, D. and Goues, C.L.: Overfitting in Semantics-based Automated Program Repair, *Empirical Software Engineering*, Vol.23, No.5, pp.3007–3033 (2018).
[33] Wang, S., Wen, M., Lin, B., Wu, H., Qin, Y., Zou, D., Mao, X. and Jin, H.: Automated Patch Correctness Assessment: How Far Are We?, *Proc. ASE '20*, pp.968–980 (2020).
[34] Yang, J., Zhikhartsev, A., Liu, Y. and Tan, L.: Better Test Cases for Better Automated Program Repair, *Proc. ESEC/FSE '17*, pp.831–841 (2017).
[35] Yu, Z., Martinez, M., Danglot, B., Durieux, T. and Monperrus, M.: Test Case Generation for Program Repair: A Study of Feasibility and Effectiveness, arXiv: 1703.00198 (2017).
[36] Xin, Q. and Reiss, S.P.: Identifying Test-Suite-Overfitted Patches through Test Case Generation, *Proc. ISSTA '17*, pp.226–236 (2017).
[37] Assiri, F.Y. and Bieman, J.M.: Fault localization for automated program repair: Effectiveness, performance, repair correctness, *Software Quality Journal*, Vol.25, No.1, pp.171–199 (2017).
[38] Liu, K., Koyuncu, A., Bissyandé, T.F., Kim, D., Klein, J. and Traon, Y.L.: You Cannot Fix What You Cannot Find! An Investigation of Fault Localization Bias in Benchmarking Automated Program Repair Systems, *Proc. ICST '19*, pp.102–113 (2019).
[39] Liu, K., Wang, S., Koyuncu, A., Kim, K., Bissyandé, T.F., Kim, D., Wu, P., Klein, J., Mao, X. and Traon, Y.L.: On the Efficiency of Test Suite based Program Repair: A Systematic Assessment of 16 Automated Repair Systems for Java Programs, *Proc. ICSE '20* (2020).
[40] Sousa, M., Dillig, I. and Lahiri, S.K.: Verified Three-Way Program Merge, *Proc. ACM on Programming Languages*, Vol.2, No.OOPSLA (2018).
[41] Leßenich, O., Apel, S. and Lengauer, C.: Balancing precision and performance in structured merge, *Automated Software Engineering*, Vol.22, No.3, pp.367–397 (2015).
[42] Accioly, P., Borba, P., Silva, L. and Cavalcanti, G.: Analyzing Conflict Predictors in Open-Source Java Projects, *Proc. MSR '18*, pp.576–586 (2018).
[43] Accioly, P., Borba, P. and Cavalcanti, G.: Understanding Semi-Structured Merge Conflict Characteristics in Open-Source Java Projects, *Empirical Software Engineering*, Vol.23, No.4, pp.2051–2085 (2018).
[44] Shen, B., Zhang, W., Zhao, H., Liang, G., Jin, Z. and Wang, Q.: IntelliMerge: A Refactoring-Aware Software Merging Technique, *Proc. ACM on Programming Languages*, Vol.3 (2019).
[45] Pastore, F., Mariani, L. and Micucci, D.: BDCI: Behavioral Driven Conflict Identification, *Proc. ESEC/DSE '17*, pp.570–581 (2017).
[46] Westfechtel, B.: Structure-Oriented Merging of Revi-

sions of Software Documents, *Proc. SCM '91*, pp.68–79 (1991).
[47] Buffenbarger, J.: Syntactic Software Merging, *Proc. SCM '94 and SCM '95*, pp.153–172 (1995).
[48] Niu, N., Easterbrook, S. and Sabetzadeh, M.: A Category-Theoretic Approach to Syntactic Software Merging, *Proc. ICSM '05*, pp.197–206 (2005).
[49] 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).
[50] Apel, S., Leßenich, O. and Lengauer, C.: Structured Merge with Auto-Tuning: Balancing Precision and Performance, *Proc. ASE '12*, pp.120–129 (2012).
[51] Cavalcanti, G., Borba, P. and Accioly, P.: Evaluating and Improving Semistructured Merge, *Proc. ACM on Programming Languages*, Vol.3 (2017).
[52] Lippe, E. and van Oosterom, N.: Operation-based Merging, *SIGSOFT Softw. Eng. Notes*, Vol.17, No.5, pp.78–87 (1992).
[53] Dig, D., Manzoor, K., Johnson, R.E. and Nguyen, T.N.: Effective Software Merging in the Presence of Object-Oriented Refactorings, *IEEE Trans. Softw. Eng.*, Vol.34, No.3, pp.321–335 (2008).
[54] 西村雄一，紙名哲生，丸山勝久：コードの編集履歴を用いた競合解決支援，情報処理学会論文誌，Vol.59, No.4, pp.1120–1136 (2018).
[55] Horwitz, S., Prins, J. and Reps, T.: Integrating Noninterfering Versions of Programs, *ACM Trans. Prog. Lang. Syst.*, Vol.11, No.3, pp.345–387 (1989).
[56] 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).
[57] Ernst, M.D., Cockrell, J., Griswold, W.G. and Notkin, D.: Dynamically Discovering Likely Program Invariants to Support Program Evolution, *IEEE Trans. Softw. Eng.*, Vol.27, No.2, pp.99–123 (2001).
[58] Nguyen, H.V., Nguyen, M.H., Dang, S.C., Kästner, C. and Nguyen, T.N.: Detecting Semantic Merge Conflicts with Variability-Aware Execution, *Proc. ESEC/FSE '15*, pp.926–929 (2015).
[59] Pacheco, C., Lahiri, S.K., Ernst, M.D. and Ball, T.: Feedback-Directed Random Test Generation, *Proc. ICSE '07*, pp.75–84 (2007).
[60] Fraser, G. and Arcuri, A.: EvoSuite: Automatic Test Suite Generation for Object-Oriented Software, *Proc. ESEC/FSE '11*, pp.416–419 (2011).
[61] Goues, C.L., Nguyen, T., Forrest, S. and Weimer, W.: GenProg: A Generic Method for Automatic Software Repair, *IEEE Trans. Softw. Eng.*, Vol.38, No.1, pp.54–72 (2011).
[62] Martinez, M. and Monperrus, M.: ASTOR: A Program Repair Library for Java, *Proc. ISSTA '16*, pp.441–444 (2016).
[63] 柗本真佑，肥後芳樹，有馬 諒，谷門照斗，内藤圭吾，松尾裕幸，松本淳之介，富田裕也，華山魁生，楠本真二：高処理効率性と高可搬性を備えた自動プログラム修正システムの開発と評価，情報処理学会論文誌，Vol.61, No.4, pp.830–841 (2020).
[64] Weimer, W., Fry, Z.P. and Forrest, S.: Leveraging Program Equivalence for Adaptive Program Repair: Models and First Results, *Proc. ASE '13*, pp.356–366 (2013).
[65] Qi, Y., Mao, X., Lei, Y., Dai, Z. and Wang, C.: The Strength of Random Search on Automated Program Repair, *Proc. ICSE '14*, pp.254–265 (2014).
[66] Le, X.B.D., Lo, D. and Goues, C.L.: History Driven Program Repair, *Proc. SANER '16*, pp.213–224 (2016).
[67] Kim, D., Nam, J., Song, J. and Kim, S.: Automatic Patch Generation Learned from Human-Written Patches, *Proc. ICSE '13*, pp.802–811 (2013).
[68] Long, F. and Rinard, M.: Staged Program Repair with Condition Synthesis, *Proc. ESEC/FSE '15*, pp.166–178 (2015).
[69] Xiong, Y., Wang, J., Yan, R., Zhang, J., Han, S., Huang, G. and Zhang, L.: Precise Condition Synthesis for Program Repair, *Proc. ICSE '17*, pp.416–426 (2017).
[70] Saha, R.K., Lyu, Y., Yoshida, H. and Prasad, M.R.: ELIXIR: Effective Object Oriented Program Repair, *Proc. ASE '17*, pp.648–659 (2017).
[71] Hua, J., Zhang, M., Wang, K. and Khurshid, S.: Towards Practical Program Repair with On-demand Candidate Generation, *Proc. ICSE '18*, pp.12–23 (2018).
[72] Yuan, Y. and Banzhaf, W.: ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming, *IEEE Trans. Softw. Eng.*, Vol.46, No.10, pp.1040–1067 (2020).
[73] Le, X.-B.D., Chu, D.-H., Lo, D., Goues, C.L. and Visser, W.: JFIX: Semantics-based Repair of Java Programs via Symbolic PathFinder, *Proc. ISSTA '17*, pp.376–379 (2017).
[74] Long, F., Amidon, P. and Rinard, M.: Automatic Inference of Code Transforms for Patch Generation, *Proc. ESEC/FSE '17*, pp.727–739 (2017).

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 年より立命館大学理工学部助教授．2007 年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．

邢 小茜

2019 年 8 月立命館大学大学院理工学研究科博士課程前期課程修了．在学中は，自動プログラム修正の研究に従事．現在，MetaApp（Beijing）に勤務．