自動プログラム修正によるマージ競合の自動解決を目指して

丸山 勝久[1,a)]　邢 小茜[2]

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，競合するコード片を特定し，それらのコード片を混ぜ合わせたソースコードがテストケースを満たすように修正しなければならない．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，テストによって検出されたバグを自動的に修正する自動プログラム修正技法を活用することで，マージ競合を自動的に解決する手法を提案する．さらに，実験により，マージ後のソースコードが自動生成できる可能性を示す．提案手法を導入することにより，マージ競合の解決において開発者の負担を軽減することができる．

1. はじめに

近年のソフトウェア開発では，それぞれの開発者が独立してソースプログラム[*1]の変更を行い，変更の完了後にそれらをマージするのが一般的になりつつある．このような並行開発により複数の開発者が効率的に作業を行うことが可能となり，ソフトウェア開発の生産性が高まる [1,2]．

並行開発では，それぞれの開発者が相手側の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ることがある [2]．この重なり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 [3]．マージ競合の存在が，効率的な並行分散開発を促進する上での障壁となっている [4]．

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，(3) 予期しない振る舞いを実行する振る舞い競合に分類できる [3]．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おいて発生し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，振る舞い競合を検出することも可能である．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，それらを矛盾なく混ぜ合わせることを指す．

マージ競合の解決における開発者の負担を軽減するために，半自動でマージ競合を解決する技法が提案されている．たとえば，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，マージを試みる手法を提案している [5]．また，Apel らは，マージを試みる開発者が，プログラムの構造を考慮しない行単位のマージを行うのか，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ている [6,7]．しかしながら，これらの手法は，構文的競合や静的な意味的競合の解決を目指しており，振る舞い競合の解決を対象としていない．これに対して，Marcelo らは，`git-merge` や `kdiff3` で採用されている 3-way マージにおいて振る舞い競合を発生させない条件を示し，その条件を検査するツールを提案している [8]．残念ながら，このツールは競合の検出を目的としており，その解決は対象としていない．このような状況において，振る舞い競合の解決作業は，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っているのが現状であり，開発者にとって大きな負担である．

本論文では，マージ対象のソースプログラムに対して，ソフトウェア移植（software transplantation）[9] の考え方を適用することで，振る舞い競合を自動的に解決する手法を提案する．このための手段として，我々は，自動プログラム修正（以下 APR と呼ぶ）技法 [10] に着目した．APR とは，バグ（欠陥）を含むプログラムとその振る舞いに関

[1] 立命館大学情報理工学部
[2] 立命館大学大学院情報理工学研究科
[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．

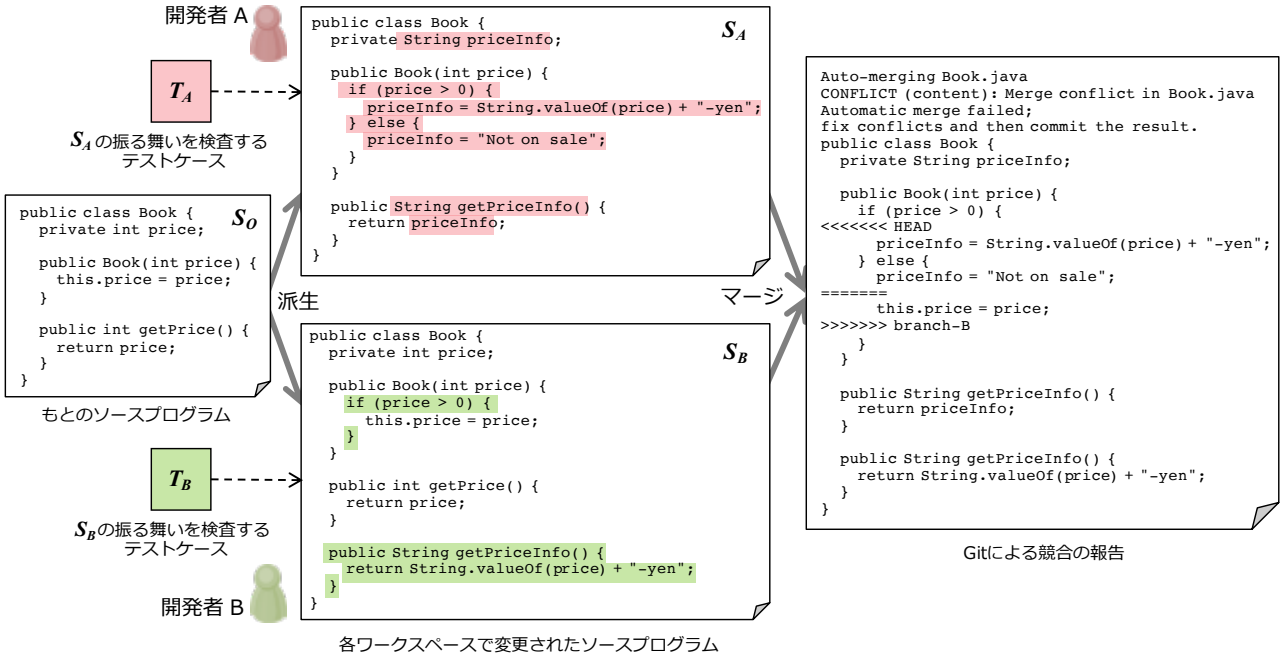

図 1 マージ競合の例

する正解を与えて，バグを取り除くパッチを出力する技術である．提案手法では，バグを含むプログラムに対して，それ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生成&検証（generate-and-validate）戦略 [11] を活用する．この戦略では，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みなす．

我々は，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 [12] に基づき，遺伝的プログラミングの概念を取り入れた kGenProg [13] を APR システムとして採用した．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 [14]．さらに，将来的な改変を意識して設計されているため，改造が容易である．

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，マージ競合（バグ）を含む（可能性のある）初期ソースプログラムを作成する．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を利用することが特徴である．我々の実施した実験では，振る舞い競合の半自動的な解決に成功した．

我々の知る限り，本手法がマージ競合の解決に対する APR の最初の適用である．提案手法の基本的な発想は，International Workshop on Intelligent Bug Fixing で発表済みである [15]．これに対して，本論文で提案する手法では，自動マージの実現可能性を改善するために，初期ソースプログラムの作成手法の改良と，APR システムとして採用した kGenProg の改造を行った．さらに，実験に用いる例題を追加し，提案手法の実現可能性に関する評価を行った．

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る舞い競合を自動的に解決するために，APR を活用した自動マージ手法を提案する．
- マージ競合の解決に APR を適用するために，修正対象のソースプログラムを機械的に作成する仕組みと修正材料を選択する仕組みを述べる．
- 実験を通して，振る舞い競合の自動解決が実現できることを示す．

以下，2 章ではマージ競合の例を示す．3 章では APR を活用した提案手法を説明する．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

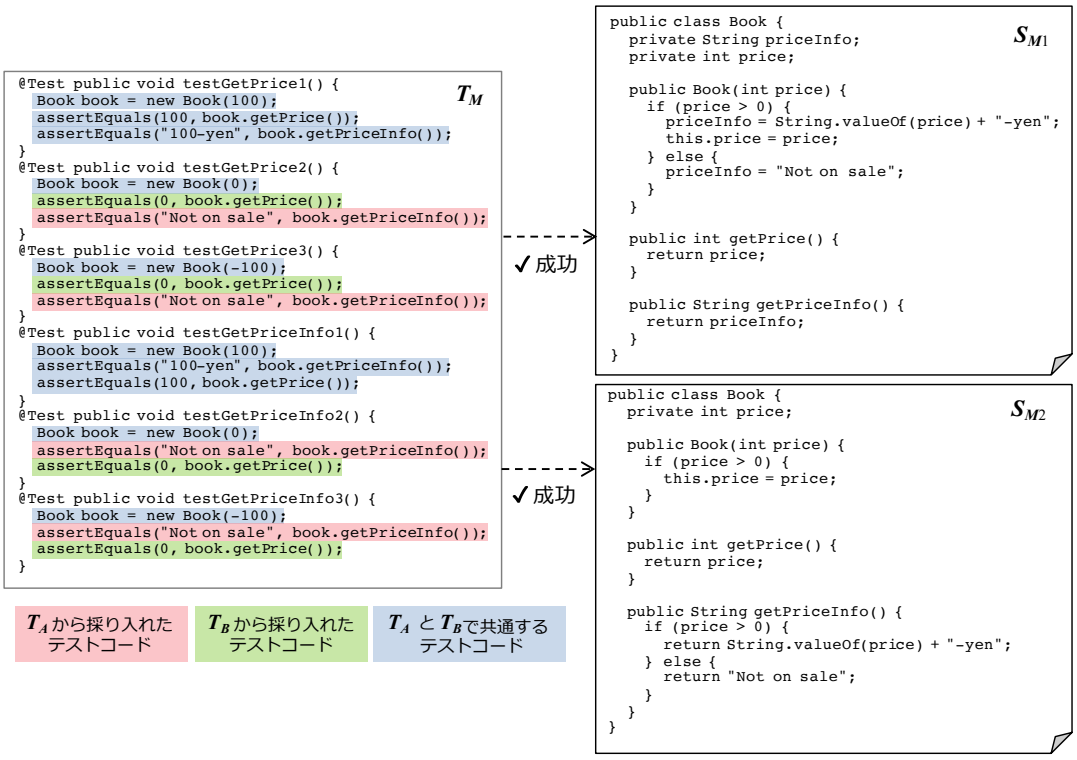

図 2 マージ競合を検査するテストスイートと手動マージにより生成したソースプログラム

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() の振る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，2 つのソースプログラム S_{M1} と S_{M2} 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．いま，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，このような冗長に見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_{M2} 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 の振る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，一見冗長に見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．よって，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，人手による振る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

遺伝的アルゴリズムに基づく APR システムでは，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．提案手法において，テストケースがあらかじめ用意されていることを前提としている．このようなテストケースは，振る舞い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．

これに対して，コンパイル可能な修正対象のソースプロ

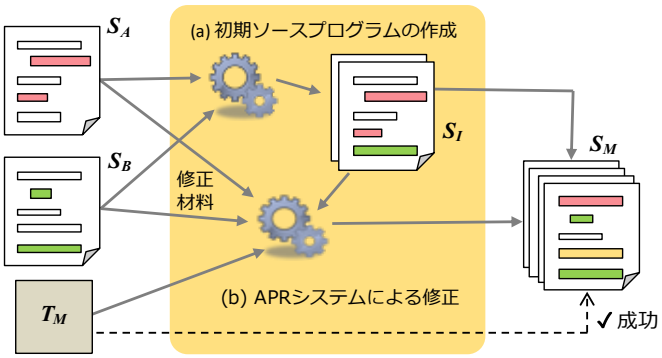

図 3　提案手法の概要

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を加工して利用する．さらに，修正材料には，マージ対象のソースプログラム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，Menezes らの調査結果 [16] 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．

上記に述べた方針に基づく提案手法の概要を図 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る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 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 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_I を修正する．T_M を満たす S_M はそのまま S_I とする．

以下，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 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．

いま，S_A と S_B に含まれるクラス `Book` のクラスメンバ `Book`$[S_A]$ と `Book`$[S_B]$ を次のように集合で表現する．

`Book`$[S_A] = \{ m_{A1}, m_{A2}, m_{A3} \}$
`Book`$[S_B] = \{ m_{B1}, m_{B2}, m_{B3}, m_{B4} \}$
m_{A1}: フィールド `priceInfo`
m_{A2}: メソッド `Book()`
m_{A3}: メソッド `getPriceInfo()`
m_{B1}: フィールド `price`
m_{B2}: メソッド `Book()`
m_{B3}: メソッド `getPrice()`
m_{B4}: メソッド `getPriceInfo()`

通常，APR はメソッド本体の修正を対象としており，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困難である．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，提案手法では，できるだけ数多くのクラスメンバを初期ソースプログラムに含む方針をとる．

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み合わせを考えると，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，提案手法では，以下の方針により組み合わせの数を削減する．

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_A と m_B を考える．m_A と m_B は同一のシグネチャを持つ．m_A と m_B の両方が書き換えられている（m_A と m_B と同じシグネチャを持つ C_O のクラスメンバ m_O が存在し，m_O，m_A，m_B が互いに異なる）場合，あるいは両方が新規に追加された（m_A と m_B のどちらも C_O に存在しない）場合，m_A と m_B を排他的関係と定義する．この関係を $m_A|m_B$ と表現する．これに対して，m_A と m_B のどちらか一方のみが書き換えられている場合，m_A と m_B を排他的関係と捉えず，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_A と m_B 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，テスト実行により，このようなクラスメンバを特定することは容易である．

上記の方針を `Book`$[S_A]$ と `Book`$[S_B]$ に適用すると，S_I に含まれるクラスメンバ `Book`$[S_I]$ は以下のようになる．

`Book`$[S_I] = \{$
$m_{A1}, m_{B1}, m_{A2}|m_{B2}, m_{B3}, m_{A3}|m_{B4} \}$

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については，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，そのまま挿入される．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 つのクラスが生成できる．

$$\texttt{Book}[S_{I1}] = \{\ m_{A1}, m_{B1}, m_{A2}, m_{B3}, m_{A3}\ \}$$
$$\texttt{Book}[S_{I2}] = \{\ m_{A1}, m_{B1}, m_{B2}, m_{B3}, m_{B4}\ \}$$
$$\texttt{Book}[S_{I3}] = \{\ m_{A1}, m_{B1}, m_{A2}, m_{B3}, m_{B4}\ \}$$
$$\texttt{Book}[S_{I4}] = \{\ m_{A1}, m_{B1}, m_{B2}, m_{B3}, m_{A3}\ \}$$

これらのクラスをそれぞれ含む 4 つの初期ソースプログラム S_{I1}，S_{I2}，S_{I2}，S_{I4} が生成される．このようにして生成された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，提案手法は，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排除する．

3.2 APR による初期ソースプログラムの修正

多くの場合，クラスメンバの組み合わせによって作成した初期ソースプログラム S_I は，マージ後のソースプログラム S_M の振る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_I を APR における修正対象とし，T_M を満たす S_M を出力する．APR システムとして kGenProg（1.5.5 版）[*2] を採用した．

ここで，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，提案手法では，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 に対して，修正材料を指定するオプションを追加する改造を行った．修正材料となるソースプログラムは構文解析が可能であればよく，クラス名が重複している（静的な意味エラーを含む）ソースプログラムでもそのまま指定可能である．

また，我々は，Menezes らの調査結果 [16] より，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感じた．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 CapGen を提案している [17]．残念ながら，我々が採用した（本手法を考案した時点の）kGenProg は文レベルの変異操作しか実現していない．そこで，我々は，条件式の置換による変異操作を追加するように，kGenProg を拡張した．

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** ）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．

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

4. 実験

提案手法が自動的にマージ競合を解決できる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　図 1 の例題を用いた実験

実験 B　複数のソースファイルがマージに関係する例題を用いた実験

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

すべての実験は，3.7GH Quad-Core Intel Xeon E5 CPU とメモリ容量 64GB を搭載した Mac Pro（macOS 10.15）で実施した．Java VM には JRE 11.0.6 を利用し，VM に 16GB のメモリ容量を割り当てた．初期ソースプログラムの作成は手動で行った．

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 の実行においては，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，すべての実験の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．実験データを `https://www.fse.cs.ritsumei.ac.jp/APR-based-Merge` 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 の結果を示す．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 は一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．本実験では，これら 2 つの値を一括して 20, 50, 100 に設定した．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

表 1 実験 A の結果

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

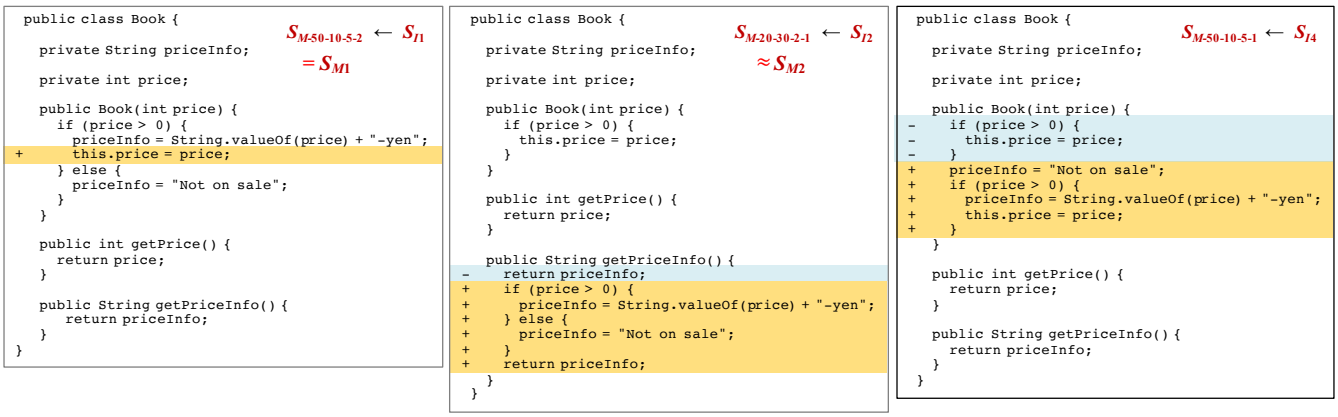

図 4 提案手法によって出力されたマージ後のソースプログラム

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 回の起動において生成にかかる合計時間である．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く，あらかじめ正解を用意することは難しい．そこで，本実験では，可読性の観点から，参照されない変数への代入文，検査が不要な（必ず真になる）条件式を持つ **if** 文，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まないソースプログラムを受け入れ可能と判断した．この作業のために，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機械的に選別した．実験 A において，選別後のソースプログラムは，$V = 20, 50, 100$ に対して，それぞれの 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，第一著者が目視で最終判断を行った．

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自動マージにより出力されている．出力されたマージ後のソースプログラムの一部を**図 4** に示す．+ は追加された行，− は削除された行を指す．ここでは，$V = v, G = g$, 乱数のシード $= r$ において出力される n 番目のソースプログラムを $S_{M-v-g-r-n}$ と表す．$S_{M-50-10-5-2}$，$S_{M-20-30-2-1}$，$S_{M-50-10-5-1}$ は，それぞれ S_{I1}，S_{I2}，S_{I4} を修正したものである．あらかじめ用意した S_{M1} に同等なソースプログラム（図 4 の左側）は，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_{M2} 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 の中央）しか出力できなかった．また，あらかじめ想定していなかったソースプログラム（たとえば，図 4 の右側）が，V の値を増やすことで 2 個出力できた．さらに，表 1 を見ると，G の値の増加は受け入れ可能なソースプラグラムの増加に貢献していない．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，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選別（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 つのソースプログラムを手に入れることができた．

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 S_I のプロジェクトに限定した（改良前の kGenProg を利用した）場合，マージ後のソースプログラムはひと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提案手法が有効であることが確認できた．

4.2 実験 B

実験 B では，複数のファイルをマージする際の競合の解決を試みる．この実験で利用する 3 つの例題（VS1, VS2, DT）を**表 2** に示す．VS1 と VS2 では，Fowler の書籍 [18] 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．S_O は変更前のソースプログラム，S_A と

表 2　実験 B におけるマージ競合

ID	競合ファイル/メソッド	競合行数	S_O（変更前）	S_A（マスターリポジトリ）	S_B（ローカルリポジトリ）
VS1	`Customer.java` `statement()`	7	L=425	リファクタリング L[S_O]=+313−35	画面表示の変更 L[S_O]=+57−10
VS2	`Customer.java` `statement()`	7	L=425	リファクタリング L[S_O]=+315−44	ポイント計算の変更 L[S_O]=+45−14
DT	`DrawCanvas.java` 複数のメソッド†	27+10	L=2,610	図形の移動機能の追加 L[S_O]=+479−24	図形の消去機能の追加 L[S_O]=+168−6

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

表 3　実験 B の結果

ID	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

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．表 2 において，競合ファイル/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 箇所で競合が報告された．L はソースプログラム S_O（テストファイルも含む）の行数，L[S_O] 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 の結果を示す．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 では，初期ソースプログラムを 2 つ作成したが，そのうちのひと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る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，本手法では，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ひと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選別（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，提案手法を利用することで，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 における変更が S_A のリファクタリングによる外部的振る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一著者が目視で確認して，受け入れの可否を判断した．もっとも簡潔な変更（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提案手法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．一般的に，テストが成功するように条件文を融合する作業は人間にとって面倒である．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 37 行（27 行と 10 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 行を越えている．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，提案手法の適用を試みた．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）．これらを第一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選別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一著者が人手で実施した．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，提案手法はマージ競合を自動的に解決できる可能性があることを示した．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．これにより，修正材料の選び方の有効性も示せた．さらに，提案手法では，kGenProg に拡張を行うことで，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ひと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まだ未着手であるが，条件式の置換による変異操作の導入という提案手法の拡張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．

一方で，マージ競合の解決において変更の規模が小さい場合（たとえば，VS1），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かになった．開発者がマージ後のソースプログラムを確認するという点から，受け入れ不可なソースプログラムを自動で選別できる仕組みを提案手法に組み込む必要がある．さらに，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．しかしながら，正箇所を Git により報告される競合箇所に限定してしまうと，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．DT のときに人手で行ったように，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重要である．

4.3 実験 C

実験 C では，実際の Java プロジェクトにおいて，提案手法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

[*3] https://github.com/rjust/defects4j
[*4] JacksonDatabind プロジェクトと JacksonXml プロジェクトは

表 4　実験 C におけるマージ競合

ID	競合ファイル/メソッド	競合行数	S_O	S_A	S_B	S_M（マージ後）
Jsoup-1	QueryParser.java byTag()	3	Lf=356	Lf[S_O]=+12−6	Lf[S_O]=+2−2	Lf[S_O]=+12−6 Lf[S_A]=+2−2, Lf[S_B]=+12−6
Jsoup-7	HttpConnection.java createConnection()	11	Lf=807	Lf[S_O]=+90−3	Lf[S_O]=+25−23	Lf[S_O]=+113−24 Lf[S_A]=+25−23, Lf[S_B]=+90−3

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，ひと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（正解とみなす）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，提案手法による解決は見込めない．

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スイート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，これらは振る舞い競合である．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，提案手法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，提案手法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 プロジェクトの 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．競合ファイル/メソッドと競合行数は，表 4 と同じである．Lf は（プロジェクトのソースプログラム全体ではなく）変更前ソースコード S_O に存在する競合ファイル（競合する前のソースファイル）の行数，Lf[S] はソースプログラム S に存在する競合ファイルに対する追加行数（+）と削除行数（−）を指す．

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．

[*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 の結果

競合	V	S_M	S_U	T(秒)	判定
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

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踏まえ，打ち切りの世代数は 30 とした．**表 5** に実験 C 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．

本実験では，プロジェクトにおいて受け入れられた（正解とみなせる）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，提案手法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．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る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，提案手法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，マージ対象のソースプログラム S_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ているにもかかわらず，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 については，S_A に対して追加および削除行数がそれぞれ 25 行と 23 行であっても，equiv の出力までに 6,240 秒（104 分）を費やしている．マージ競合の解決を夜間のバッチ処理で実施するとい

う用途であれば許容できる可能性が残されている．しかしながら，マージ競合の解決を統合開発環境上でリアルタイムに支援するという立場からは許容できる実行時間とはいえない．実用性の観点から，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求められる．

4.4 妥当性の脅威

実験 B では，汎用的な例題を用意した．しかしながら，著者らが例題の作成と提案手法の設計の両方を実施しているため，例題が提案手法に有利になっている可能性がある．また，それぞれの実験において，生成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一著者の主観である．判定者によっては，結果が変わる恐れがある．

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．また，実験 B の DT および実験 C における修正箇所の範囲と修正材料の指定は，提案手法の能力に対する評価を歪めている可能性がある．さらに，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．

5. 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マージ競合の解決手法を提案した．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生成に成功した．一方で，実際のソフトウェア開発に導入するためには，受け入れ不可なソースプログラムを選別する仕組みの導入と実行時間の短縮が必要である．残念ながら，4.3 節で示したように，現実のプロジェクトにおいて，提案手法が適用できる場面はまだ少ない．構文的競合や静的な意味的競合の解決も考慮して，適用場面を広げるための改良を予定している．

提案手法では，自動プログラム修正システムとして，kGenProg を改造して利用しているが，テストスイートに基づく他の APR [19] を利用することも検討している．また，提案手法の基本的な発想は，生成&検証戦略の APR での利用を前提としたものではない．精度および時間効率の観点で，JFix [20] のような合成ベース技法の活用も検討している．

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頂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, *CACM*, Vol. 52, No. 9, pp. 56–62 (2009).
[2] Bird, C. and Zimmermann, T.: Assessing the Value of Branches with What-If Analysis, *Proc. FSE '12*, pp. 45:1–11 (2012).
[3] Mens, T.: A State-of-the-Art Survey on Software Merging, *IEEE TSE*, Vol. 28, No. 5, pp. 449–462 (2002).
[4] Phillips, S., Sillito, J. and Walker, R.: Branching and Merging: An Investigation into Current Version Control Practices, *Proc. CHASE '11*, pp. 9–15 (2011).
[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] 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).
[8] Sousa, M., Dillig, I. and Lahiri, S. K.: Verified Three-Way Program Merge, Vol. 2, No. OOPSLA (2018).
[9] Barr, E. T., Harman, M., Jia, Y., Marginean, A. and Petke, J.: Automated Software Transplantation, *Proc. ISSTA 2015*, pp. 257–269 (2015).
[10] Gazzola, L., Micucci, D. and Mariani, L.: Automatic Software Repair: A Survey, *IEEE TSE*, Vol. 45, No. 1, pp. 34–67 (2019).
[11] Weimer, W., Nguyen, T., Goues, C. L. and Forrest, S.: Automatically Finding Patches Using Genetic Programming, *Proc. ICSE '09*, pp. 364–374 (2009).
[12] 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, *EMSE*, Vol. 22, No. 4, pp. 1936–1964 (2017).
[13] 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).
[14] 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).
[15] Xing, X. and Maruyama, K.: Automatic Software Merging using Automated Program Repair, *Proc IBF '19*, pp. 11–16 (2019).
[16] Menezes, G. G. L., Murta, L. G. P., Barros, M. O. and van Der Hoek, A.: On the Nature of Merge Conflicts: a Study of 2,731 Open Source Java Projects Hosted by GitHub (2018).
[17] 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).
[18] Fowler, M.: *Refactoring: Improving the Design of Existing Code*, Addison-Wesley (1999).
[19] Liu, K., Wang, S., Koyuncu, A., Kim, K., Bissyandé, T. F. D. A., 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).
[20] 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).