推薦論文● FOSE2020(特集●ソフトウェア工学の基礎)

コード記述パターンに基づく素材コード片生成による自動プログラム修正手法

安田 和矢　伊藤 信治　中村 知倫　原田 真雄　肥後 芳樹

近年，ソースコードのデバッグ作業を効率化する自動プログラム修正が注目されている．従来の自動プログラム修正手法では，欠陥が含まれる命令を，ソースコード中の別の箇所に出現する式 (素材コード片) を用いて書き換えることで，欠陥を修正する．したがって，必要な素材コード片がソースコード中に存在しない場合，欠陥を修正できない．本研究では，修正対象システムで特定のコード記述パターンがよく用いられる，という開発者の知見を活用し，事前定義したコード記述パターンに基づき素材コード片を生成する手法を提案する．これにより，ソースコード中に素材コード片が存在しない場合でも，プログラムの自動修正が可能となる．実製品の開発中に検出・修正された欠陥 48 件に提案手法を適用した結果，既存手法では修正できた欠陥の数が 9 件だったところ，提案手法では 2 件増加し 11 件となった．

In recent years, automated program repair techniques have attracted attention. In the conventional techniques, a defect is fixed by rewriting statements containing the defect using expressions (donor code) that appear in another part of the source code. If donor code required to fix the defect does not exist in the source code, the conventional techniques cannot repair the defect. In this study, a technique is proposed, which fixes defects even in the above-mentioned case. In the proposed technique, developers' knowledge that specific expressions are often used in the system is specified as the commonly used expression pattern in advance, and expressions commonly used in the system are generated as donor code according to the pattern. As a result of applying the proposed technique to 48 defects detected and fixed during the development of an actual product, the number of defects that could be fixed by the conventional technique was nine, and the number increased by two to eleven in the proposed technique.

1 はじめに

IT 人材不足が深刻化する近年において，デバッグ作業の効率化を目的とした多くの研究が進められている．経済産業省の試算によれば，2018 年時点で約 22 万人の IT 人材が不足しており，2030 年には 45 万人程度まで人材の不足規模が拡大すると見込まれている [10]．IT 人材不足が進む中，需要の増加に対応するためには，開発生産性をこれまで以上に高める必要がある．特に，システムの実装・テストに必要なコストの 50% をデバッグ作業が占めており [3]，この作業の効率化を図る研究が盛んに進められている．

ソースコードのデバッグ作業を効率化する技術として，自動プログラム修正が提案されている [7] [11] [13] [15]．自動プログラム修正は，欠陥を含むソースコードとテストスイートを入力として，ソースコードからその欠陥を取り除く技術である．自動プログラム修正の手法として，Generate-and-Validate (G&V) と呼ばれる手法が知られている．G&V では，ソースコード中から欠陥が含まれる命令を特定した後，与えられた全てのテストケースが成功するようになるまでソースコードを書き換える．従来の G&V では，ソースコードの書き換え時に，ソースコード中に存在する式を修正用のコード片 (素材コード片) として使用する．事前に定義したルール (修正パター

Automated Program Repair Using Donor Code Generation Based on Features of Targeted Systems.

Kazuya Yasuda, Shinji Itoh, Tomonori Nakamura, Masao Harada, 株式会社日立製作所, Hitachi, Ltd.

Yoshiki Higo, 大阪大学大学院情報科学研究科, Osaka University.

コンピュータソフトウェア, Vol.38, No.4 (2021), pp.23–32.

[研究論文] 2021 年 2 月 10 日受付.

ン) にしたがって，欠陥が含まれる命令の一部分を素材コード片に変更したり，欠陥が含まれる命令の前後に素材コード片を挿入したりすることで，ソースコードを書き換えていく．

従来の G&V の手法には，欠陥の修正に必要な素材コード片がソースコード中に存在しない場合，その欠陥を修正できないという問題がある．この問題は，ソースコードを 1 度だけ書き換える手法 [12] だけでなく，遺伝的アルゴリズムによりソースコードを複数回書き換える手法 [11] [15] でも生じる．これらの手法は，欠陥の修正に必要な素材コード片は既にソースコード中に存在している，という仮説に基づいている [14]．しかし，Barr らによる調査 [2] では，オープンソースのプロジェクトにおける変更履歴 (コミット) のうち，ソースコード中に存在するような行から生成可能なコミットは 43% しか存在しないことが示されている．より多くの欠陥を修正するためには，ソースコード中に存在しない式を素材コード片として利用する必要がある．

本研究では，修正に必要な素材コード片がソースコード中に存在しない場合であっても，修正対象システムに対する開発者の知見を活用し，必要な素材コード片を新たに生成する自動プログラム修正手法を提案する．提案手法では，修正対象システムのソースコードを記述する上でよく利用される式が存在する，という開発者の知見を事前に「頻出式パターン」として定義する．頻出式パターンは，式の型 (クラス) ごとに，その型を持つ式が別の式の中でどのように利用されるか，という修正対象システムにおいて頻出するコード記述パターンを定義する．欠陥の修正時，頻出式パターンに基づき，ソースコード中に存在する式からよく利用される式を素材コード片として生成することで，修正可能な欠陥を増やすことを試みる．本手法では，例えば Data Transfer Object (DTO) パターンのようなデザインパターンを採用するシステムや，ヘルパークラスやユーティリティクラスを持つシステムなど，ある型を持つ式に対して，その式を含むようなコード記述パターンを定義可能な場合に有用である．

本稿では，エンタープライズシステムにおいて DTO パターンがよく用いられるという特徴に注目した頻出式パターンの一例を示し，これを用いた評価実験の結果を示す．DTO に対してはソースコードにはゲッターメソッド呼び出しをよく利用することから，ゲッターメソッド呼び出し式を頻出式パターンとして定義した．本手法を，実製品の開発中に検出・修正された欠陥 48 件に適用した結果，既存手法では修正できた欠陥の数が 9 件だったところ，提案手法では 2 件増加し 11 件となった．

2 自動プログラム修正

本章では，自動プログラム修正の概要と，本研究の対象とする問題について述べる．様々な自動プログラム修正手法が提案されているが，本稿ではその中でもよく知られている Generate-and-Validate (G&V) という種類の手法について，既存手法 TBar [12] を例に説明する．

2.1 概要

自動プログラム修正とは，1 つ以上の欠陥を含むソースコードと，失敗するテストケースを含むテストスイートを入力として，与えられた全てのテストケースに成功するソースコードを出力する技術である．自動プログラム修正は主に， 欠陥箇所特定とパッチ生成の 2 つの処理で構成される．欠陥箇所特定では，ソースコードから欠陥を含む可能性 (疑惑値) の高い命令を特定する．パッチ生成では，疑惑値の高い命令を書き換えることで，与えられた全てのテストケースに成功するソースコードを生成する．

多くの自動プログラム修正手法では，Spectrum-Based Fault Localization と呼ばれる欠陥箇所特定の方法が用いられる [16]．この方法では，各テストケースにおける各命令の実行有無とテストの成否の情報に基づき，各命令に対して疑惑値を算出する．疑惑値の算出方法としては Ochiai [1] などが使用される．

図 1 に，TBar のパッチ生成処理の概要を示す．パッチ生成では，パッチ候補生成とパッチ候補検証を繰り返す．パッチ候補生成では，疑惑値の高い命令から 1 つを選択し，当該命令を事前に定義したルール (修正パターン) に従って書き換え，複数のパッチ候補を出力する．パッチ候補検証では，出力された各

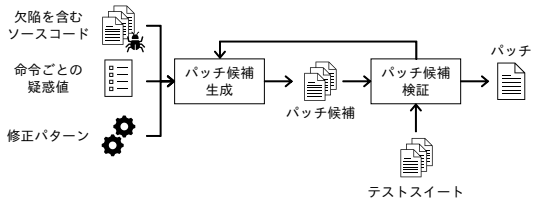


図 1 TBar のパッチ生成処理

パッチ候補を元のソースコードに適用した上で，テストを実行する．全てのテストケースに成功するパッチ候補が得られた場合，それを最終的なパッチとして出力する．そうでない場合には，パッチ候補生成からやり直す．

パッチ候補生成時に書き換えの対象とする命令を選ぶ方法は，手法によって異なる．TBar では，疑惑値の値が大きい命令から順番に書き換えの対象とする．全てのテストケースに成功するパッチ候補が 1 つでも生成された場合，その時点でパッチ生成を終了する．そうでない場合には，生成したパッチ候補を破棄し，元のソースコードで疑惑値の値が次に大きい命令を書き換えの対象として，パッチ生成を繰り返す．

パッチ候補生成で用いる修正パターンとして，命令の挿入・置換・削除といった基本的な操作 (Atomic Change Operators) のみを定義する手法や，より複雑な操作 (Template-Based Change Operators) を定義する手法が存在する．TBar は後者の手法で，15 の修正パターンを定義してパッチ候補生成に用いる．表 1 に，これら修正パターンの一覧を示す．演算子の変更 (Mutate Operators) のような基本的な操作や，配列外参照をチェックする条件分岐の挿入 (Insert Range Checker) のような，欠陥の修正時に頻出する操作が定義されている．

図 2 に，修正パターンを用いたパッチ候補生成の例を示す．ここでは “Mutate Variable” (疑惑値の高い命令に含まれる変数を別の式に書き換える) という修正パターンを例として用いる．

パッチ候補生成では，まず，ソースコード中に出現する式を全て取得する．以降では，ここで取得した式を素材コード片 (Donor Code) と呼ぶ．図 2 の例では，素材コード片として変数 `info` と式 `getRInfo(info)` が取得されている．素材コード片の探索範囲としては，書き換えの対象とする命令を含むファイル (クラス) や，そのクラスを含むパッケージ全体，システム全体などが考えられる．TBar では，書き換えの対象とする命令を含むクラスのみを探索範囲とする．

表 1 TBar で用いられる修正パターン [12]

#	修正パターン名
1	Insert Cast Checker
2	Insert Null Pointer Checker
3	Insert Range Checker
4	Insert Missed Statement
5	Mutate Class Instance Creation
6	Mutate Conditional Expression
7	Mutate Data Type
8	Mutate Integer Division Operation
9	Mutate Literal Expression
10	Mutate Method Invocation Expression
11	Mutate Operators
12	Mutate Return Statement
13	Mutate Variable
14	Move Statement
15	Remove Buggy Statement

次に，疑惑値の高い命令を修正パターンに従って書き換え，パッチ候補を生成する．ここで，書き換え後に用いる式として，先ほど取得した素材コード片が使用される．図 2 に示す例では，疑惑値の高い命令に含まれる変数 `filename` が，素材コード片 `info` や `getRInfo(info)` に書き換えられている．なお，書き換えにおいては，素材コード片のうち書き換え箇所と型が一致する式のみを用いる．

2.2 G&V の問題

G&V の手法では，ソースコード中に出現する式を素材コード片として，疑惑値の高い命令の一部分を素材コード片に書き換えることで欠陥の修正を図る．したがって，欠陥を正しく修正するために必要な素材コード片がソースコード中に存在しない場合，その欠陥を修正できない．例えば，図 2 において，変数 `filename` を別の式 `info.getTitle()` に書き換えなければ不具合を修正できない場合を考える．このとき，式 `info.getTitle()` がソースコード中に存在しないと，素材コード片として取得されないため，この欠陥を修正できない．

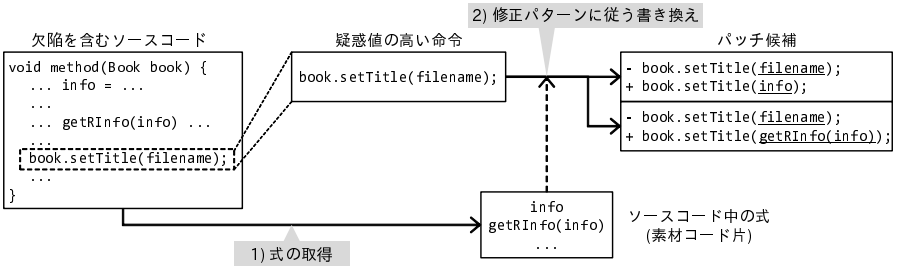


図 2　修正パターンを用いたパッチ候補生成の例

3　提案手法

本章では，前述の G&V の問題を解決する手法 Temp-Don (Template-Based Program Repair Using Donor Code Generation) を提案する．Temp-Don は，修正対象システムに対する開発者の知見を活用し，修正に必要となる素材コード片を生成することで，より多くの欠陥に対応する手法である．

3.1　概要

Temp-Don は，修正パターンとして複雑な操作 (Template-Based Change Operators) を採用する既存手法 TBar をベースとした手法である．事前定義したコード記述パターン (頻出式パターン) に従って素材コード片の生成 (Donor Code Generation) を行い，パッチ候補生成に用いる．

図 3 に，提案手法におけるパッチ候補生成の概要を，図 2 と同様の例を用いて示す．提案手法では，以下の 3 ステップによってパッチ候補生成を行う．

ステップ 1　従来のパッチ候補生成と同様に，ソースコード中の式を取得する．素材コード片の探索範囲は，書き換えの対象とする命令を含むクラスのみとする．

ステップ 2　「頻出式パターン」に従って，ソースコード中の式から別の式を合成することで，素材コード片を生成する．頻出式パターンの詳細については後述する．

ステップ 3　従来のパッチ候補生成と同様に，疑惑値の高い命令を修正パターンに従って書き換え，パッチ候補を生成する．

3.2　頻出するコード記述パターンの定式化

Temp-Don は，修正対象システムのソースコードを記述する上でよく利用される式が存在する，という開発者の知見を利用するため，頻出する式を「頻出式パターン」として定式化する．頻出式パターンでは，式の型 (クラス) ごとに，その型を持つ式が別の式の中でどのように利用されるかを定義する．以下に，拡張バッカス・ナウア記法 (EBNF) [8] による頻出式パターン *pattern* の構文規則を示す (簡略化のため，トークンの区切りを示す空白文字は省略する)．

$$\begin{aligned} \mathit{pattern} &= \mathit{params},\ \text{“:”},\ \mathit{expression};\\ \mathit{params} &= \{\ \mathit{param},\ \text{“,”}\ \},\ \mathit{param};\\ \mathit{param} &= \mathit{classname},\ \mathit{var};\end{aligned}$$

ここで，*classname* は型，*var* はその型の式を表すプレースホルダであり，*expression* は，修正対象システムのプログラミング言語における任意の式である．*expression* には，*params* で定義したプレースホルダを用いて，修正対象システムでよく利用される式を記述する．頻出式パターン *pattern* は複数定義できる．

図 3 に，頻出式パターンと，それに従い生成される素材コード片の例を示している．図 3 には，2 つの頻出式パターンが定義されている．1 つ目の頻出式パターンは，型 `Info` を持つ任意の式 `x` が，メソッド呼び出し式 `x.getTitle()` として利用されることを示している．2 つ目は，型 `Info` を持つ任意の式 `x` が，メソッド呼び出し式 `x.getSubTitle()` として利用されることを示している．

Temp-Don は頻出式パターンに従いソースコード中に存在する式から別の式を合成することで，素材コード片を生成する．図 3 の例では，ソースコード中の

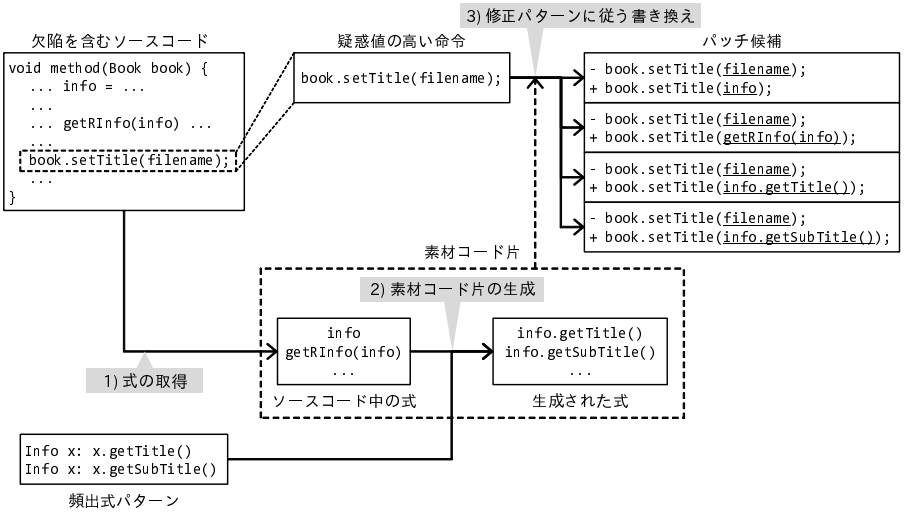


図 3 提案手法におけるパッチ候補生成

`Info` 型の式 `info` について，1 つ目の頻出式パターンに従い素材コード片 `info.getTitle()` を，2 つ目の頻出式パターンに従い素材コード片 `info.getSubTitle()` を生成している．

頻出式パターンを用いると，ある式に着目したときに，その式が修正対象システムのソースコードでどのように利用されることが多いか，という開発者の知見を記述できる．頻出式パターンでは，型を持つプレースホルダを用いて，頻出式パターンを定義する．静的型付きプログラミング言語においては，よく利用されるメソッド呼び出し式やフィールド参照式は型を用いて抽象化することができるため，提案手法における頻出式パターンを用いてよく利用されるメソッド呼び出し式やフィールド参照式を記述できる．

例えば，ヘルパークラスやユーティリティクラスがシステム中に存在する場合，そのクラスのメソッド呼び出し式はよく利用されると考えられる．ヘルパークラス $ClassH$ のクラスメソッド $method$ に対して，$ClassX$ 型のオブジェクトを引数にとるとき，以下の通り頻出式パターンを記述することで，ヘルパークラスのメソッド呼び出し式を素材コード片として生成できる．

$$ClassX\ x \quad : \quad ClassH.method(x)$$

また，システムが特定のデザインパターンを採用している場合，そのパターンで定義されたメソッド呼び出し式はよく利用されると考えられる．例えば Visitor パターン [6] においては，Visitor クラス $VisitorX$ のメソッド $visit$ や，Acceptor クラス $AcceptorY$ のメソッド $accept$ が定義される．以下の通り頻出式パターンを記述することで，これらメソッド呼び出し式を素材コード片として生成できる．

$$VisitorX\ x, AcceptorY\ y \quad : \quad x.visit(y)$$
$$VisitorX\ x, AcceptorY\ y \quad : \quad y.accept(x)$$

複数の命令列が決まった順番で呼び出される場合や，特定のアルゴリズムが頻出する場合など，よく利用される単位が式ではなく命令文である場合，頻出式パターンではその命令文を記述できない．この場合，一連の命令列やアルゴリズムを，頻出式パターンではなく修正パターンとして定義することで，よく利用される命令文をパッチ候補として生成できる．

前述のように，ある型を持つ式に対して，その式を含むようなコード記述パターンを定義可能な場合に，本手法は有用である．以降ではその一例として，エンタープライズシステムでよく用いられるデザインパターン「DTO パターン」に注目し，このデザインパターンで頻出するゲッターメソッド呼び出し式を素材コード片として生成する手法を示す．

3.3 DTO に注目した頻出式パターン

Data Transfer Object (DTO) は，クラス間 (特に，ビジネスロジック層のコンポーネント間や，ビジネスロジック層とプレゼンテーション層の間) でのデータ交換を行う際に用いられるオブジェクトである [5]．図 4 に，DTO クラスの例を示す．DTO として渡されたデータを使用する際には，ゲッターメソッドを呼び出すことで，格納された値を取り出す．J2EE[†1] デザインパターンでは「DTO パターン」として紹介されており [5]，DTO はエンタープライズシステムでよく用いられている．

DTO パターンを利用するエンタープライズシステムのソースコードには，ゲッターメソッド呼び出しが頻出する．そこで本稿では，修正対象システムのクラス *ClassX* に含まれるゲッターメソッド *getY* ごとに，以下の通り頻出式パターンを定義する．

$$ClassX\ x : x.getY()$$

この頻出式パターンは，ソースコードを解析することで，自動的に生成できる．

疑惑値の高い命令で参照可能な DTO は，ソースコード中から取得した式に含まれる．上記のように頻出式パターンを定義することで，提案手法によって自動修正可能な欠陥を増やすことができると考えられる．

4 評価

4.1 実装

本研究では，欠陥の修正成功率を評価するため，提案手法 Temp-Don をプロトタイプ実装した．実装したプロトタイプは，Java[†2] 言語で記述されたソースコードを対象としている．欠陥箇所特定には GZoltar [4] を用い，疑惑値の算出方法として Ochiai を使用する．頻出式パターンとしては，前章で述べた，ゲッターメソッド呼び出し式の記述を使用した．

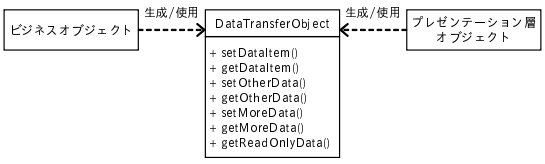


図 4 Data Transfer Object クラスの例 [5]

†1 J2EE は Oracle America, Inc. の米国における登録商標．

†2 Java は Oracle America, Inc. の登録商標．

素材コード片生成の効果を評価するため，提案手法との比較用のベースラインとして，既存手法 TBar もプロトタイプ実装した．Temp-Don との違いは，素材コード片の生成処理を行わない点のみである．

4.2 評価題材

評価題材として，企業で開発された実システムで，開発途中のある期間に検出・修正された欠陥 48 件を用いた．題材システムは Java 言語で実装されており，プロダクトコードは 250 クラス，約 35,000 LoC (Lines of Code) からなる．開発途中の 1 年間で検出・修正された欠陥の中から，テストによって検出可能で，ソースコードのみの変更により修正可能な欠陥 48 件を抽出した．開発者によって作成された欠陥の情報 (バグ票) を元に，著者らが題材システムの開発リポジトリから上記 48 件の欠陥を再現し，評価に用いた．

4.3 評価方法

評価題材の欠陥 48 件に対して，提案手法 Temp-Don と既存手法 TBar のプロトタイプをそれぞれ実行し，欠陥を修正できたか否かを確認した．プロトタイプの実行は，提案手法と既存手法ともに，12 個の論理 CPU と 16GB の RAM をもつ仮想マシン上で行った．

全てのテストケースに成功するパッチが生成されても，そのパッチが題材システムの開発者によって実装された修正内容と等しいとは限らない．本評価では開発者によって実装された修正内容と意味的に等価なパッチが，プロトタイプによって出力されたパッチに含まれているか否かによって，欠陥を修正できたか否か判断した．

4.4 結果

表 2 に，Temp-Don と TBar のプロトタイプを実行した結果を示す．「パッチ出力」には，各手法によっ

て，全てのテストケースに成功するパッチが出力された欠陥の件数を示す．「パッチ未出力」には，各手法によって，全てのテストケースに成功するパッチが 1 つも出力されなかった件数を示す．パッチが出力された欠陥件数のうち，「開発者と等価」には，開発者によって実装された修正内容と等価なパッチが含まれていた件数を示す．「開発者と相違」には，そのようなパッチが 1 つも含まれていなかった件数を示す．

表 2 に示す通り，全てのテストケースに成功するパッチを出力できた欠陥の件数は，TBar では 9 件 (= 8 + 1) だったところ，Temp-Don では 11 件 (= 9 + 2) となった．開発者による修正内容と等価なパッチが含まれていた欠陥の件数と，等価なパッチが含まれていなかった欠陥の件数は，いずれも Temp-Don によって 1 件ずつ増加した．

表 3 に，Temp-Don と TBar で実行結果が異なった 3 つの欠陥について，各手法の実行結果を示す．表 3 に示す通り，TBar ではパッチを出力できなかった 2 つの欠陥において，Temp-Don で開発者による修正内容と等価なパッチが生成された．一方で，1 つの欠陥において，TBar では開発者による修正内容と等価なパッチを生成できていたが，Temp-Don では同様のパッチを出力できず，開発者による修正内容と異なるパッチのみが出力された．なお，TBar で開発者と等価なパッチを出力できた残りの欠陥 7 件においては，Temp-Don でも全く同じパッチが出力された．

表 2 各手法の実行結果と該当する欠陥の件数

実行結果	Temp-Don	TBar
パッチ出力 (開発者と等価)	9	8
パッチ出力 (開発者と相違)	2	1
パッチ未出力	37	39
合計	48	48

表 3 各手法で実行結果が異なった欠陥

ID	Temp-Don	TBar
1	パッチ出力 (開発者と等価)	パッチ未出力
2	パッチ出力 (開発者と等価)	パッチ未出力
3	パッチ出力 (開発者と相違)	パッチ出力 (開発者と等価)

表 4 各手法で欠陥の修正に要した時間

	Temp-Don	TBar
平均値	1 時間 40 分 51 秒	32 分 37 秒
中央値	24 分 54 秒	11 分 59 秒
最小値	27 秒	15 秒
最大値	7 時間 56 分 32 秒	1 時間 35 分 04 秒

表 4 に，Temp-Don と TBar のプロトタイプが欠陥の修正に要した時間を示す．表には，各手法でパッチ出力できた欠陥 (Temp-Don では 11 件，TBar では 9 件) のみを抜粋し，平均値，中央値，最小値，最大値を集計した結果を示している．

表 4 に示す通り，Temp-Don では TBar と比較して，平均値で約 3 倍，中央値で約 2 倍の時間を修正に要した．

4.5 考察

表 2 に示した通り，提案手法では既存手法と比較して，パッチを出力できた欠陥の数が 9 件 (= 8 + 1) から 11 件 (= 9 + 2) に増加した．また，表 3 に示した通り，既存手法ではパッチを出力できなかった 2 つの欠陥において，提案手法では開発者による修正内容と等価なパッチを出力できるようになった．提案手法によってゲッターメソッド呼び出し式を素材コード片として生成したことで，既存手法より多くの欠陥を修正できるようになったと考えられる．

図 5 に，TBar では開発者による修正と等価なパッチを出力できなかったが，Temp-Don では出力できた欠陥と，出力されたパッチの例を示す．図 5 には，表 3 の ID が 1 の欠陥を示しており，メソッド呼び出し式の引数 `aaaInfo.getInfoID()` が誤っている．ただし，図 5 に示すコードは，変数名やメソッド名などが実際のコードとは異なる．

図 5 に示すパッチ例は，“Mutate Method Invo-

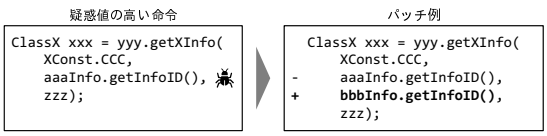


図 5 提案手法により出力されたパッチ例

cation Expression” (疑惑値の高い命令に含まれるメソッド呼び出し式を書き換える) という修正パターンにより生成されたパッチである．この修正パターンでは，メソッド呼び出し式の引数が，素材コード片に書き換えられる．この欠陥を含むソースコード中には修正に必要な式 `bbbInfo.getInfoID()` が含まれないため，TBar ではパッチ出力できなかった．一方 Temp-Don では，ソースコード中に存在する変数 `bbbInfo` にメソッド呼び出し式を付け加え，素材コード片として `bbbInfo.getInfoID()` を生成したため，図 5 に示すパッチを出力できた．

表 3 に示した通り，1 つの欠陥について，既存手法では開発者による修正と等価なパッチを出力できていたが，提案手法では開発者による修正と異なるパッチが出力された．ここで出力されたパッチは，当該欠陥の修正としては正しくないと考えられるパッチだった．これは，提案手法によって既存手法より多くのパッチ候補が生成されたことが原因であると考えられる．全てのテストケースに成功するパッチが生成されれば，開発者による修正内容と等価か否かに関わらず，そのパッチが出力される．したがって，多くのパッチ候補が生成されることで，開発者による修正内容とは異なるパッチも多く出力されるようになったと考えられる．特に，提案手法の入力とするテストスイートが，通常の開発においては十分であっても，提案手法で生成されるような，正しくないパッチ候補の識別には不十分な場合，正しくないパッチが出力されると考えられる．

提案手法 Temp-Don を既存手法 TBar と組み合わせて実行することで，上述の欠陥でも開発者による修正と等価なパッチを出力できると考えられる．具体的には，最初に TBar を実行し，一定時間内にパッチ出力できなかった場合，疑惑値の最も高い命令から Temp-Don を実行し直す，という方式が考えられる．この方式を用いることで ID 3 の欠陥でも開発者による修正と等価なパッチを出力できる．一方，ID 1, 2 のような Temp-Don のみでパッチ出力できる欠陥に対しては，TBar の実行終了を待つ必要があるため，修正までに多くの時間を要するという問題点がある．

表 4 に示した通り，提案手法では既存手法と比較して，欠陥の修正に平均で約 3 倍の時間を要した．しかし，最も時間の要する欠陥でも 8 時間以内に修正できており，CI 環境での夜間実行などの使い方においては，十分実用的な修正時間であると考えられる．

5 関連研究

従来の G&V による自動プログラム修正の研究では，修正可能な欠陥を増やすために，パッチの探索空間を拡大した上で，その空間を効率よく探索するための手法が数多く提案されている．

GenProg [11] は，遺伝的アルゴリズムを用いてパッチを探索する．修正パターンとしては命令単位での操作 (挿入・置換・削除) のみを採用するが，遺伝的アルゴリズムを用いることで，操作を複数組み合わせたパッチを探索できる．

ただし GenProg では，命令単位の操作だけで実現不可能なパッチを出力できない．GenProg に，提案手法の素材コード片生成を組み合わせることで，命令単位の操作だけでは実現できないパッチ候補も生成できるようになり，より多くの欠陥を修正できると考えられる．

CapGen [15] は， 命令単位での操作に加えて，式の単位での操作まで修正パターンに加えた手法である．操作のバリエーションが増え，パッチ候補の探索空間が大きくなるため，コンテキストを考慮に入れた素材コード片の優先度付けによって効率よく探索することをねらっている．

LSRepair [13] は，修正対象のシステムだけでなく，別システムのソースコードからも素材コード片を取得する手法である．膨大な量のソースコードから適切な素材コード片を見つけるために，修正するメソッドと素材コード片を含むメソッドの類似度よって，素材コード片の優先度付けを行う．

一方，提案手法では，修正対象のシステムで特定の式がよく用いられることが明らかな場合に，その式のみを素材コード片として生成するため，パッチの探索空間が抑えられる．したがって，提案手法は CapGen や LSRepair と比較して，探索空間の拡大を抑えることができ，効率よく欠陥を修正できると考えられる．

6 おわりに

本研究は，G&V による自動プログラム修正において，修正可能な欠陥を増やすことを目的とする．本研究では，修正対象システムのソースコードを記述する上でよく用いられる式が存在する場合に，そのような式を頻出式パターンとして事前に定式化し，そのパターンに従って素材コード片を生成する手法を提案した．本稿ではその一例として，エンタープライズシステムで DTO パターンがよく用いられるという特徴に注目し，頻出するゲッターメソッド呼び出し式を頻出式パターンとして定義した．提案手法を実製品の開発中に検出・修正された欠陥 48 件に適用した結果，既存手法では修正できた欠陥の数が 9 件だったところ，提案手法では 2 件増加し 11 件となった．

今後の課題として，提案手法を，TBar 以外の既存手法と比較することが挙げられる．GenProg などのメタヒューリスティクスを用いる手法と比較し，提案手法では探索空間が抑えられると考えられる．一方，探索空間や探索効率，修正に要する時間について実験的に確認できていないため，今後はこれらの評価を進める．

より多くの題材での評価も今後の課題である．Defects4J [9] のようなオープンソースの評価題材では，DTO が用いられておらず，提案手法の評価に適さない．企業における他の実製品の題材を用意したうえで，追加評価を行っていく．

参 考 文 献

[1] Abreu, R., Zoeteweij, P., and van Gemund, A. J. C.: On the Accuracy of Spectrum-based Fault Localization, in *Proceedings of the Testing: Academic and Industrial Conference - Practice And Research Techniques*, 2007, pp. 89–98.

[2] Barr, E. T., Brun, Y., Devanbu, P., Harman, M., and Sarro, F.: The Plastic Surgery Hypothesis, in *Proceedings of the 22nd International Symposium on Foundations of Software Engineering*, 2014, pp. 306–317.

[3] Britton, T., Jeng, L., Carver, G., Cheak, P., and Katzenellenbogen, T.: Reversible Debugging Software, University of Cambridge Judge Business School, 2013.

[4] Campos, J., Riboira, A., Perez, A., and Abreu, R.: GZoltar: An Eclipse Plug-In for Testing and Debugging, in *Proceedings of the 27th International Conference on Automated Software Engineering*, 2012, pp. 378–381.

[5] Crawford, W. and Kaplan, J.: *J2EE Design Patterns*, O'Reilly Media, 2003. [邦訳] 有限会社福龍興業, 佐藤直生 (監訳), 木下哲也 (訳): J2EE デザインパターン, オライリージャパン, 2004.

[6] Gamma, E., Helm, R., Johnson, R., and Vlissides, J.: *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, 1994.

[7] Gazzola, L., Micucci, D., and Mariani, L.: Automatic Software Repair: A Survey, *IEEE Trans. Softw. Eng.*, Vol. 45, No. 1 (2019), pp. 34–67.

[8] ISO/IEC 14977:1996: Information Technology – Syntactic Metalanguage – Extended BNF.

[9] Just, R., Jalali, D., and Ernst, M. D.: Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs, in *Proceedings of the International Symposium on Software Testing and Analysis*, 2014, pp. 437–440.

[10] 経済産業省: IT 人材需給に関する調査, 2019.

[11] Le Goues, C., Nguyen, T., Forrest, S., and Weimer, W.: GenProg: A Generic Method for Automatic Software Repair, *IEEE Trans. Softw. Eng.*, Vol. 38, No. 1 (2012), pp. 54–72.

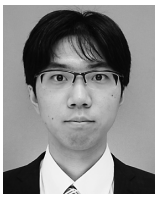
[12] Liu, K., Koyuncu, A., Kim, D., and Bissyandé, T. F.: TBar: Revisiting Template-Based Automated Program Repair, in *Proceedings of the 28th International Symposium on Software Testing and Analysis*, 2019, pp. 31–42.

[13] Liu, K., Koyuncu, A., Kim, K., Kim, D., and Bissyandé, T. F.: LSRepair: Live Search of Fix Ingredients for Automated Program Repair, in *Proceedings of the 25th Asia-Pacific Software Engineering Conference*, 2018, pp. 658–662.

[14] Weimer, W., Nguyen, T., Le Goues, C., and Forrest, S.: Automatically Finding Patches Using Genetic Programming, in *Proceedings of the 31st International Conference on Software Engineering*, 2009, pp. 364–374.

[15] Wen, M., Chen, J., Wu, R., Hao, D., and Cheung, S.-C.: Context-Aware Patch Generation for Better Automated Program Repair, in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 1–11.

[16] Wong, W. E., Gao, R., Li, Y., Abreu, R., and Wotawa, F.: A Survey on Software Fault Localization, *IEEE Trans. Softw. Eng.*, Vol. 42, No. 8 (2016), pp. 707–740.



安田和矢

2014 年東京大学理学部情報科学科卒業．2016 年同大学大学院情報理工学系研究科コンピュータ科学専攻修士課程修了．同年株式会社日立製作所入社．ソースコード解析，形式検証などの研究開発に従事．IEEE 会員．

伊藤信治

1998 年九州大学工学部電気工学科卒業．2000 年同大学大学院システム情報科学研究科修了．同年株式会社日立製作所入社．要求工学，形式手法，ソフトウェア開発プロセスなどの研究開発に従事．情報処理学会正会員．

中村知倫

2003 年電気通信大学情報通信工学科卒業．2005 年同大学大学院情報通信工学専攻修了．同年株式会社日立製作所入社．現在，アプリケーション開発効率化のための各種ソリューション開発に従事．

原田真雄

2003 年明治大学理工学部電気電子工学科卒業．同年日立ソフトウェアエンジニアリング株式会社入社．2015 年株式会社日立製作所へ承継入．現在，ソース自動修正やソースコード診断支援などのシステム開発に従事．

肥後芳樹

2002 年大阪大学基礎工学部情報科学科中退．2006 年同大学大学院博士後期課程修了．2007 年同大学院情報科学研究科コンピュータサイエンス専攻助教．2015 年同准教授．博士 (情報科学)．ソースコード解析，特に自動プログラム修正やコードクローン検出に関する研究に従事．情報処理学会，電子情報通信学会，日本ソフトウェア科学会，IEEE 各会員．