

共変更されるソースコード修正パターンの抽出

福元 春輝†
Haruki Fukumoto

伊原 彰紀‡
Akinori Ihara

1. はじめに

ソフトウェアの機能拡張や改善のために実装されたソースコードは、検証作業（コードレビュー）によって高品質なソフトウェアを実現している。一般的なコードレビューでは、複数人が対面の会議において検証対象の仕様書やソースコードを目視で確認し、欠陥の検出、ソースコードの可読性、などを確認する。昨今では、オンラインサービスの普及にともないソースコードを GitHub で管理するようになり、対面で実施していたコードレビューが、オンライン環境で実施するコードレビュー（モダンコードレビュー）の形式に移行している。モダンコードレビューは、対面による作業に比べ効率的になっているが、コードレビューは未だ検証と再修正を繰り返すコストのかかる作業となっている [1][2][3]。

Bacchelli らは、コードレビューの目的について開発者 873 人にインタビューを行い、337 人 (39%) の開発者から、可読性の改善などのソースコードの品質向上が目的であると回答を得た [4]。レビューで行われている議論においても、75% はソフトウェアの保守性に関するものである [5][6]。本研究は、コードレビューにおけるソースコードの改善コスト削減方法の一つとして、ソースコードの自動修正の研究に取り組む。

ソースコードの自動修正の研究は、これまでも多数提案され、自動修正の方法は 2 つに分類できる。1 つは検索ベースの自動修正方法、もう 1 つはテンプレートベースの自動修正方法である。検索ベースの自動修正方法として、GenProg [7] や SPR [8] などがあり、これらは検索アルゴリズムに基づき、不具合修正を行う。Goues らは、GenProg が OSS プロジェクトの 105 個の欠陥のうち、55 個を修正可能であることを明らかにし [7]、Long らは SPR を OSS プロジェクトにおいて GenProg よりも修正精度が高いことを明らかにしている [8]。しかし、検索ベースの修正手法では、開発者が受け入れないソースコードを生成することも多く [8]、GenProg が生成したソースコードの 94.33%、SPR が生成したソースコードの 70% は開発者に受け入れられないことを明らかにした [7]。一方で、テンプレートベースの自動修正方法として、PAR [9] が提案されている。Kim らは、PAR によって修正したソースコードは、GenProg よりも多くのソースコードを修正できることを明らかにしている [9]。しかし、テンプレートベースの自動修正方法は、修正前後のプログラムを正規表現などでテンプレート化した修正パターンに基づく自動修正をするため、1 行単位での修正パターンを対象としている。

ソフトウェア開発では、複数行のソースコードを同時に変更することも多く、複数のソースコードを同時に変更することを考慮して修正パターンを作成する必要がある。本研究は、コードレビューにおける変更を対象に、テンプレートベースの自動修正方法として、共変更を考慮した複数

行にわたるソースコードの変更パターンを抽出する方法を提案する。具体的には、同一メソッド内に含まれる変更行間の結合関係を考慮し、ソースコードが同時変更されるパターンを抽出する。

ケーススタディとして、多数の変更依頼が投稿されている neo4j プロジェクト (GitHub における neo4j/neo4j リポジトリ) を対象に、コードレビューにおいて採択されたプルリクエスト 1,398 件に含まれる、開発者が提案した 5,214 件のコミットを対象に変更パターンを計測する。

続く 2 章では関連研究について述べ、本研究の立ち位置を説明する。3 章では、本研究での手法について説明し、4 章で実データから検出した変更パターンを示す。5 章で考察を行い、6 章で妥当性への脅威を述べる。最後に 7 章で本研究をまとめる。

2. 従来研究

2.1 ソースコード自動修正技術

Goues らは、遺伝的アルゴリズムを用いて欠陥を自動修正する GenProg を提案した [7]。GenProg は、プログラム内のソースコードの再利用と、欠陥を含むソースコードの変異を繰り返すことで欠陥修正したソースコードの生成を行う。GenProg は、8 つの OSS プロジェクトにおける 105 個の欠陥のうち 55 個を自動修正することを確認した [7]。しかし、検索ベースの自動修正技術では、検索対象内に欠陥を修正するためのソースコードが存在しない限り修正できず、また開発者が受け入れできない可読性が低いソースコードが生成されることが多いという課題もある [10]。

検索ベースの自動修正技術の課題を解決する方法として、テンプレートベースの自動修正技術を用いた PAR が提案されている [8]。テンプレートベースの修正技術は、過去のソースコード変更履歴から修正パターンを作成し、修正パターンに合致するソースコードを自動修正する。Kim らは、5 つの OSS プロジェクトにおいて、PAR が GenProg よりも修正精度が高いことを示した [8]。テンプレートベースの修正技術では、可読性が高く、精度の高い修正を行うことができる一方で、修正前後のプログラムを正規表現などでテンプレート化した修正パターンに基づく自動修正するため、1 行単位での修正のみ可能である。

2.2 プログラム内の結合関係

ソフトウェアは、複数行にわたるソースコードを組み合わせてることによって機能を実装しており、ソースコード間には多数の依存関係がある。ソースコード間の依存関係を表す指標として、結合度などがある [11]。したがって、1 行のソースコードを変更することで依存関係のある別の行に影響を与えることも少なくない [12]。

2.3 本研究における仮説

検索ベースのソースコード自動修正技術は、複数行にわたるソースコードの同時変更が可能であるが、開発者が受け入れできないソースコードを生成することも多く、採用するソースコードの選択に膨大なコストがかかる。一方で、

†和歌山大学大学院, Graduate School, Wakayama University

‡和歌山大学, Wakayama University

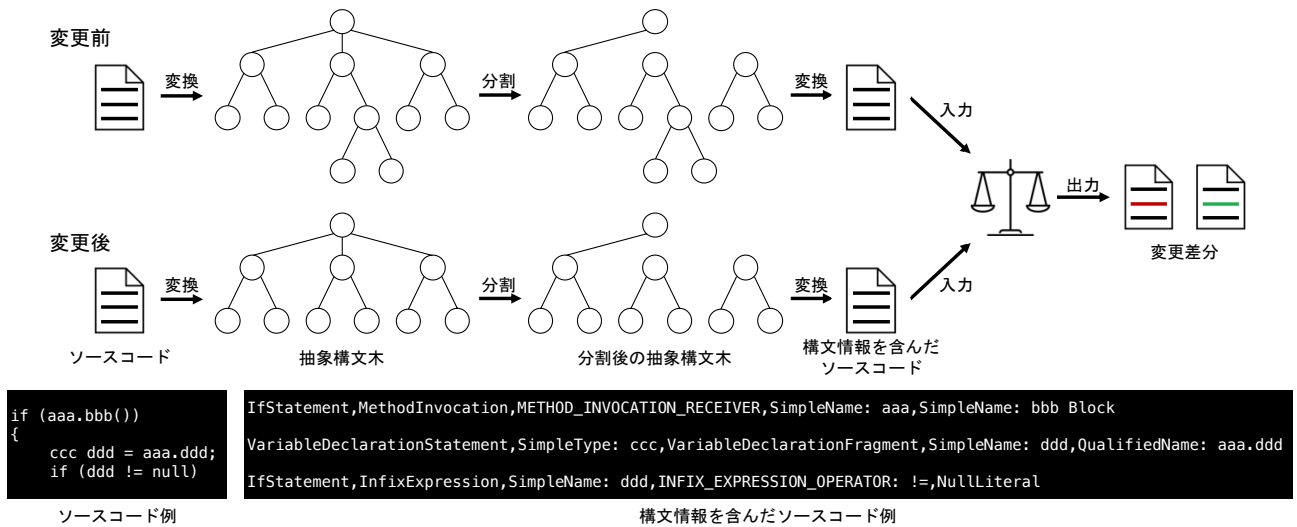


図1 変更差分検出手法の概要

変更前ソースコード

変更後ソースコード

1	try	1	try
2	{	2	{
3	if (test())	3	string str = input();
4	{	4	if (str != null)
5	string str = input();	5	{
6	if (str != null)	6	write(str);
7	{	7	}

図2 diffコマンド実行結果

テンプレートベースの自動修正技術は、変更規模が小規模である。本研究では、コストを削減するという利点を残した状態で、規模の大きい自動修正を実現するため、共変更するソースコードの変更パターン抽出方法を提案する。

3. 提案手法

本研究では、共変更する変更パターンを抽出方法として、ソースコード変更履歴から、修正パターンの検出、抽出、抽象化、分類の4つの手順を提案する。図1は、修正パターンの抽出手法の概略図を示す。

3.1 変更箇所の検出

ソースコードの変更パターン抽出にあたり、実装者が変更したソースコードを追跡する。ソースコード変更箇所の特定には、diffコマンドが頻繁に利用される。diffコマンドは、変更前後のソースコード間の最長共通部分、最小編集距離を求めることで変更箇所の差分を検出する。図2は、diffコマンドの実行結果を示す。変更前の4行目と変更後の5行目は、同じ{であるが、正しくは、変更前の7行目と変更後の5行目の{が対応している。diffコマンドは2つの文書間の文字列のみを比較しているため、図1のように構造的には異なるが、文字列が一致する行で、間違った差分を検出することがある。このようなソースコード変更箇所の特定におけるdiffコマンドの問題点を改善するため、GumtreeDiffが提案されている[13]。GumtreeDiffは、変更前後のソースコードをそれぞれ抽象構文木へと変換し、抽象構文木のノード比較を行うことで、プログラム構造を比較し正しく差分を検出する。ただし、GumtreeDiffであっても、変更されていない箇所にも変更が加えられたと表示

Aメソッド[n(変更行)=3]

変更前ソースコード	変更後ソースコード	変更パターン
行番号	実コード	行番号
27		1 28行目
28 - a = hoge;	28 + a = Hoge;	2 ----- 30行目
29	29	3 31行目 32行目
30	30 + b = B(a);	
31	31	4 28行目 28行目
31 - return a;	32 + return b;	5 28行目 28行目
		6 31行目 30行目
		6 31行目 32行目

図3 変更パターンの抽出例

されるなど、GumTreeDiffがソースコードスニペットの移動や更新と判断したうち55%が不正確であったことも実験的に示されている[14]。そこで、本研究では、GumtreeDiffのアルゴリズムとは別の手法でノード比較を行う。

本研究では変更箇所を行単位で捉えるために、GumtreeDiffが作成した抽象構文木をもとに、変更差分を検出する。具体的には、3つの手順で変更箇所を特定する(図1参照)。

- (1) **抽象構文木への変換** GumtreeDiffを用いて変更前後のソースコードをそれぞれ抽象構文木に変換する。
- (2) **抽象構文木の分割** 1ファイルのソースコードを1つの抽象構文木で表現しているため、抽象構文木をソースコード中の各行との対応関係を明らかにし、ソースコードを行単位の構文木に分割する。
- (3) **変更差分の特定** 構文木で表現した変更前後のソースコードを対象に差分比較を行うことで、構造を考慮した変更箇所を特定する。変更差分は3種類に分類する。
 - **ソースコード行の追加:** ソースコード行が、変更前のプログラムに存在せず、変更後のみに存在する。
 - **ソースコード行の削除:** ソースコード行が、変更後のプログラムに存在せず、変更前のみに存在する。
 - **ソースコード行の変更:** 対応するソースコード行が変更前後のプログラム両方に存在し、実装内容が一部異なっている。

3.2 変更パターンの抽出

コードレビューにおいて1ファイル中のソースコードの変更箇所は、単一行のみの変更と、複数行にわたる変更と2種類に分ける。複数行にわたる変更の場合、同時に変更さ

表 1 - ソースコードの抽象化例

	抽象化前		抽象化後	
	変更前	変更後	変更前	変更後
A	Method_AA(Method_A());	int a = Method_A(); Method_AA(a);	Method1(Argument 1);	Type1 Name1 = Method1(); Method1(Argument2);
B	Method_AA(Method_A());	int b = Method_B(); Method_AA(a);	Method1(Argument 1);	Type1 Name1 = Method1(); Method1(Argument2);
C	String str = Test.A(); int id = Test.B();	String str = Tests.A(); int id = Tests.B();	Type1 Name1 = Re1.Method1(); Type1 Name1 = Re1.Method1();	Type1 Name1 = Re2.Method1(); Type1 Name1 = Re2.Method1();
D	String str = Test.A(); int id = test.B();	String str = Tests.A(); int id = tests.B();	Type1 Name1 = Re1.Method1(); Type1 Name1 = Re1.Method1();	Type1 Name1 = Re2.Method1(); Type1 Name1 = Re2.Method1();

れた行を共変更パターンと捉える。ただし、本研究では、同一メソッド内において同時に変更された場合に限り共変更パターンとする。同一メソッド中に変更されたコード行数 n において、行単位の組み合わせから算出される変更パターン $pattern_num$ 個を抽出する。ただし、メソッド内の変更行数 n の数が増加するほど、変更パターンの組み合わせが膨大になるため、本研究では、 r を 2 として変更パターンを抽出する。

$$pattern_num = \sum_{r=1}^n nCr$$

3.3 ソースコードの抽象化

抽出した変更パターンは、開発者が宣言した識別子を含むため、抽象化する。GumtreeDiff によって作成される抽象構文木をもとに抽象化する。抽象化は、演算子や括弧などの記号を除き、変数名やメソッド名などの単語を任意の変数名に置換する。表 1 は、変更されたソースコード行とその抽象化の例を示す。A の変更パターンと、B の変更パターンを比較すると、抽象化後の変更差分が完全一致しており、A と B は同一の変更パターンであると判断できる。

しかし、抽象化前の変更差分に着目すると、A の変更後のソースコードでは宣言した変数を引数として使用し、2 つのソースコード間に依存関係があるが、B では宣言した変数と引数に依存関係はない。そのため、抽象化後の変更差分が一致している場合でも、別の変更パターンとして捉える必要がある。

3.3 変更パターンの分類

変更パターンを正しく理解するため、次の指標に基づき変更パターン进行分类する。単一行の変更、共変更ともに、(1) 抽象化後のソースコードの変更差分をもとに分類し、共変更パターンである場合は、(2) 依存関係、(3) 使用識別子を加えた 3 つの指標をもとに分類を行う。本研究では、同時に変更されたソースコード行の結合関係を明らかにするために、2 行のソースコード間の依存関係の有無を捉えるプログラム依存グラフ作成ツール TinyPDG[13]を用いる。プログラム依存グラフは、データ依存を捉え、変数宣言とその変数の使用箇所を特定する。検出された依存関係は、変数宣言と変数利用の関係を表す。実行例 1 は、TinyPDG による依存関係検出結果を示す。以下の 2 行が同時に変更されている場合、Name1 によってデータ依存関係を持つ共変更と捉える。

また、2 行のソースコード間の依存関係に加え、同一の識別子を使用している場合、一方の識別子に変更がある場

12 行目: Type1 Name1 = Method1();
15 行目: Method1(Argument1);
12→15 行目: Name1→Argument1

実行例 1 データ依存検出

合、もう一方も変更されるという仮説のもと、2 行のソースコード間の識別子に着目する。表 1 に示す C、D の変更では、抽象化後の変更パターンは同一であるが、抽象化前のソースコードでは、C は 2 行のソースコード間で Test という識別子を共通して使用しているが、D は識別子が Test と test で異なる。したがって、C のように 2 行のソースコード間で、共通する識別子が存在する場合、使用識別子が一致する共変更と捉える。

4. ケーススタディ

4.1 データセット

本研究では、GitHub に登録されているリポジトリの中で、2021 年 4 月時点でプルリクエスト数が上位 10 件のリポジトリであり、多くの従来研究で用いられている neo4j リポジトリを対象にケーススタディを行う。表 2 は、対象データセットの詳細を示す。リポジトリに含まれる 1,398 件のプルリクエストに含まれる 5,214 個のコミット間の変更箇所を抽出する。ただし、プログラム依存グラフが作成できないプログラム、メソッド全体の追加、削除を行う変更については、分析の対象外とする。

表 2 - データセット

	全データ	対象外データ	対象データ
プルリクエスト数	2,255	857	1,398
コミット数	8,422	3,208	5,214
変更ファイル数	29,978	15,779	14,199
変更行数			35,240

4.2 計測結果

本ケーススタディでは、変更依頼において検証者からのフィードバックに基づき修正されたコードレビュープロセスに含まれる全ての変更を対象とする。表 3 には、抽出した変更パターンを著者らが目視で分類した変更箇所の一部を示す。従来研究で提案された手法でも抽出できる 1 行単位の単行変更箇所 (ID:1,2) に加え、複数行にわたって変更された共変更パターン (ID:3~10) を示す。共変更パターン (ID:3~10) には、変更内容の説明 (上段)、変更前後のソースコード事例 (中段)、変更前後それぞれの一致する

表 3 - 抽出した変更内容と変更例

単一行変更パターン							
ID1	メソッドのアクセス修飾子の削除						
	public void stop() → void stop()						
ID2	メソッド名の変更						
	public void shouldReadBackPersistedIdsWhenAggressiveReuseIsSet () throws Exception → public void shouldReadBackPersistedIdsWhenAggressiveModelsSet () throws Exception						
共変更パターン（両結合関係あり）							
ID3	値を直接使用せず、変数として使用する方法へ変更						
	1	→ int coreMembers = 3;					
	2	for (int coreServerId = 0; coreServerId < 3 ; coreServerId++) → for (int coreServerId = 0; coreServerId < coreMembers ; coreServerId++)					
	前		後	coreMembers	前		後
ID4	可変長引数を配列へ変更、それに伴う変数への代入方法の変更						
	1	public int selectSlot(Value... values) → public <V> int selectSlot(V[] values, Function<V,ValueGroup> groupOf)					
	2	Value singleValue = values[0]; → ValueGroup singleGroup = groupOf.apply (values[0]);					
	前	values	後		前	1→2: values	後
共変更パターン（データ依存のみあり）							
ID5	メソッドの返り値を直接使用せず、変数として使用する方法へ変更						
	1	→ ChannelFuture channelFuture = mock(ChannelFuture.class);					
	2	when(channel.close()).thenReturn(mock(ChannelFuture.class)); → when(channel.close()).thenReturn(channelFuture);					
	前		後		前		後
ID6	仮引数の型変更、それに伴うキャスト変換の削除						
	1	IndexQuery rangeQuery(Object from, boolean fromInclusive, Object to, boolean toInclusive) → IndexQuery rangeQuery(Value from, boolean fromInclusive, Value to, boolean toInclusive)					
	2	return IndexQuery.range(0, (Number) from, fromInclusive, (Number) to, toInclusive); → return IndexQuery.range(0, from, fromInclusive, to, toInclusive);					
	前		後		前	1→2: fromInclusive, from, to, toInclusive	後
共変更パターン（使用識別子の一致のみあり）							
ID7	ある関数のメソッド呼び出しを全て削除						
	1	index.drop(); →					
	2	index.open(); →					
	前	index	後		前		後
ID8	関数名の変更						
	1	return random Value .nextIntValue(); → return random Values .nextIntValue();					
	2	return random Value .nextDigitString(); → return random Values .nextDigitString();					
	前	Value	後	Values	前		後
共変更パターン（結合関係なし）							
ID9							
	1	this.cursorFactory = cursorFactory; →					
	2	this.backToPosition = backToPosition; →					
	前		後		前		後
ID10							
	1	schemaWrite OperationsInNewTransaction (); → schemaWrite InNewTransaction ();					
	2	createIndex(statementInNewTransaction (AUTH_DISABLED)); → createIndex(newTransaction(AUTH_DISABLED));					
	前		後		前		後

使用識別子（下段左），データ依存関係（下段右）を示す。例えば，ID3 に示す変更パターンは，2 行のソースコード変更を同時に行なっている共変更パターンであり，変更内容は「値を直接使用せず，変数として使用する方法へ変更」である。また，変更後のソースコードに存在する変数 **coreMembers** が，変更された 2 行のソースコードで共通して使用され，1 行目と 2 行目のソースコードは，

coreMembers という変数を介して，データ依存関係を持っている。本研究によって抽出した（共）変更パターンを表 3 に示す 5 つのカテゴリに分類した。それぞれのカテゴリの変更について，結果をまとめる。

- (1) **単一行変更パターン**: 従来研究で抽出された変更パターンと同様，1 行のみ変更されるパターンである。ID1 のアクセス修飾子の変更や，ID2 のメソッド名の

表 4 - 共変更の発生頻度（メソッドの返り値を直接使用せず、変数として使用する方法へ変更）

	変更前	変更後	頻度
ID 5	1. 2.when(channel.close()).thenReturn(mock(ChannelFuture.class));	1.ChannelFuture channelFuture = mock(ChannelFuture.class); 2.when(channel.close()).thenReturn(channelFuture);	22
ID 11	1. 2.when(stateMachineSPI.beginTransaction(any())).thenReturn(mock(KernelTransaction.class));	1.AuthenticationResult authenticationResult = mock(AuthenticationResult.class); 2.when(machine.spi.authenticate(any())).thenReturn(authenticationResult);	8
ID 12	1. 2. for (int coreServerId = 0; coreServerId < 3; coreServerId++)	1.int coreMembers = 3; 2.for (int coreServerId = 0; coreServerId < coreMembers; coreServerId++)	5

変更のように同一メソッド内で他の行を変更する必要のない変更をパターンとして抽出した。

- (2) **共変更パターン（両結合関係あり）**：2 行のソースコード変更が同時に行われ、2 行のソースコード間にデータ依存、使用する識別子の一致が存在する変更パターンを抽出した。ID3 では数値の 3 を変数へ代入するという変更が行われると同時に、3 を変数へと置き換える変更を行なっている。値を変数へ代入する変更を加えたため、依存関係のある行が同時に変更されている。また、ID4 では、仮引数が可変長引数から配列へと変更したため、同様の変数を使用しているソースコード行も型変換に合わせて、実装方法の変更が行われている。
- (3) **共変更パターン（データ依存のみあり）**：2 行のソースコード変更が同時に行われ、2 行のソースコード間にデータ依存のみが存在する変更である。データ依存は、変数宣言と変数利用の関係である。そのため、ID5 や ID6 のように変数宣言を行なっているソースコード行に変更が加えられたことで、その利用箇所も同時に変更されている。
- (4) **共変更パターン（使用識別子の一致のみあり）**：2 行のソースコード変更が同時に行われ、2 行のソースコード間で同様の識別子が使用されている。同様の識別子が使用されていることから、ID7 や ID8 のように、一方の識別子に変更が加えられた場合、もう一方の識別子も同じように変更されている。
- (5) **共変更パターン（結合関係なし）**：2 行のソースコード変更が同時に行われたが、依存関係も使用識別子の一致もない変更である。単一行変更パターンが同時に変更された変更であり、どちらかの変更が行われない場合でも、もう一方の変更には影響が出ない変更である。

5. 考察

本研究は、データ依存関係、使用する識別子の一致の 2 種類の結合関係を考慮し、変更パターンを抽出することにより、従来研究で抽出していた単一行変更パターンに加え、共変更パターンが存在することを明らかにした。本章では、共変更パターンについて、特にテンプレートベースの修正技術における共変更パターンについて考察を行う。

データ依存関係：データ依存を持つ共変更パターンは、変数宣言行と変数使用行の 2 行で構成されている。一方のソースコード行は、もう一方のソースコード行の変更依存していることがわかる。表 3 の ID6 の共変更パターンでは、型変換とキャスト変換をそれぞれ行っており、独立して行われる変更ではない、依存関係の強い変更であるためパターンとして抽出できたと考える。ID5 の共変更パター

ンは、分析対象のブルリクエストにおいて 22 回の同様の変更を確認した。表 4 は、ID5 と同様の共変更パターンの事例と発生頻度を示す。ID11 や ID12 を別の変更として捉えたが、表 4 に示す 3 つの変更パターンは、全て「メソッドの返り値を直接使用せず、変数として使用する方法へ変更」を表している。したがって、一般化を行うことで同様の共変更を多数確認することができると考える。

使用識別子の一致：使用している識別子の一致が見られるソースコード行は、Type-2 レベルのコードクローンが多いことを確認した。したがって、一方のソースコード行に変更が加えられた場合、同じように記述しているソースコード行も同じように変更される可能性が高いと考えられる。データ依存関係とは異なり、一方の変更がもう一方の変更に影響することではなく、同様の変更を漏れなく行うために、必要な変更パターンであると考えられる。使用識別子が一致している共変更についても、データ依存関係を持つ共変更と同様、同内容の変更を多数確認することができたため、一般性を明らかにできると考える。

結合関係なし：結合関係のない場合、一方の変更内容は、もう一方の変更内容に全く関係が無く、互いの変更に影響を及ぼし合うことはない。そのため、同時に行う必要のある変更ではなく、それぞれの単一行変更が同じタイミングで行われたものであると考える。

単一行変更パターンと共変更パターンの違いは、一方の変更のみで完結しているか否かであると考えられる。また、データ依存関係を持つ場合、2 つのソースコード行の一方の変更漏れがあると欠陥となってしまうことから、使用識別子が一致している場合に比べて、同時に変更する重要度が高いことが分かる。そのため、強い結合関係を持つソースコードほど共変更を考慮する必要性があると考えられる。

6. 制約

6.1 内的妥当性

本研究では、共変更パターンを抽出する際に、メソッド内の結合関係のみを考慮し、変更差分の組み合わせを行った。モジュール間での結合関係を考慮することで、ID1 や ID2 のような変更は、別のメソッド内に影響を及ぼすため、新たな共変更パターンを抽出できると考えられる。ただし、プログラム内の全ての変更されたソースコード行の組み合わせを計測することは、コストのかかる作業となるため、メソッド内で完結する変更のみを対象とした。

6.2 外的妥当性

本研究では、neo4j リポジトリを対象にケーススタディを行った。今後多くの OSS のリポジトリを対象とし分析を行うことで、変更パターンの一般化を行うことができると考えられる。

7. 終わりに

本稿では、コードレビュープロセスで行われた変更を対象に、従来のテンプレートベースの修正技術で対象としている単一行変更パターンに加え、共変更パターンの存在を明らかにした。抽出した結果、共変更パターンは実際のコードレビュープロセス内で行われており、1行単位で変更を促すのではなく、同時に変更することを促す必要のある変更であることを明らかにした。加えて、共変更する必要があるソースコードには結合関係があることを明らかにした。今後は、共変更パターンの発生頻度を計測することで一般化を行い、従来のテンプレートベースの修正技術と精度比較を行うことで有用性を明らかにする。

謝辞

本研究は JSPS 科研費 18H03221 の助成を受けたものである。

参考文献

- [1] Baker, J., “Experts battle £192bn loss to computer bugs,” Feb. 2012. Accessed: 2015-12-07, <https://www.jbs.cam.ac.uk/insight/2012/cambridge-news-experts-battle-192-bn-loss-to-computer-bugs/>.
- [2] Britton, T., Jeng, L., Carver, G., Cheak, P. and Katzenellenbogen, T., “Reversible debugging software” Technical report, University of Cambridge, Judge Business School, 2013.
- [3] Shane, M., Yasutaka, K., Bram, A. and Ahmed, E.H., “The Impact of Code Review Coverage and Code Review Participation on Software Quality: A Case Study of the Qt, VTK, and ITK Projects”, Proceedings of the 11th Working Conference on Mining Software Repositories (MSR’14), pp.192–201 (2014).
- [4] Alberto, B. and Christian, B., “Expectations, Outcomes, and Challenges of Modern Code Review”, Proceedings of the 35th International Conference on Software Engineering (ICSE’13), pp.712–721 (2013).
- [5] Moritz, B., Alberto, B., Andy, Z. and Elmar, J., “Modern Code Reviews in Open-source Projects: Which Problems Do They Fix?”, Proceedings of the 11th Working Conference on Mining Software Repositories (MSR’14), pp.202–211 (2014).
- [6] Czerwinka, J., Greiler, M. and Tilford, J., “Code Reviews Do Not Find Bugs: How the Current Code Review Best Practice Slows Us Down”, Proceedings of the 37th International Conference on Software Engineering (ICSE’15), pp.27–28 (2015).
- [7] Le Goues, C., Dewey-Vogt, M., Forrest, S. and Wimer, W., “A systematic study of automated program repair: Fixing 55 out of 105 bugs for \$8 each”, Proceedings of the 34th International Conference on Software Engineering (ICSE’12), pp.3–13 (2012).
- [8] Long, F. and Rinard, M., “Staged Program Repair with Condition Synthesis”, Proceedings of the 8th Joint Meeting of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE’15), pp.448–458 (2015).
- [9] Kim, D., Nam, J., Song, J. and Kim, S., “Automatic patch generation learned from human-written patches”, Proceedings of the 35th International Conference on Software Engineering (ICSE’13), pp.802–811 (2013).
- [10] Qi, Z., Long, F., Achour, S. and Rinard, M., “An analysis of patch plausibility and correctness for generate-and-validate patch generation systems”, Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA’15), pp.24–36 (2015).
- [11] Pressman, R. S., “Software Engineering: A Practitioner’s Approach”, McGraw-Hill, Columbus, OH (2005).
- [12] Bohner, S. A. and Arnold, R. S., “Software Change Impact Analysis”, IEEE Computer Society Press, Los Alamitos, CA (1996).
- [13] Falleri, J., Morandat, F., Blanc, X., Martinez, M. and Monperrus, M., “Fine-grained and accurate source code differencing”, Proceedings of the 29th International Conference on Automated Software Engineering (ASE’14), pp.34–43 (2014).
- [14] Frick, V., Grassauer, T., Beck, F. and Pinzger, M., “Generating accurate and compact edit scripts using tree differencing”, Proceedings of the 30th International Conference on Software Maintenance and Evolution (ICSME’14), pp.264–274 (2018).