Vol.2021-SE-207 No.3
2021/3/1

API利用パターンを用いた自動プログラム修正手法

荒木 良仁[1]　桑原 寛明[2]　國枝 義敏[3]

概要：本論文では，API 利用パターンを用いた自動プログラム修正の手法を提案する．API 利用パターンは API の典型的な呼び出し列であり，API の誤った利用方法が原因のバグは API 利用パターンに従うことで修正できる可能性がある．一般に，自動プログラム修正は修正箇所の特定，修正候補の生成，修正候補の評価の 3 段階で構成されるが，提案手法では API 利用パターンに基づく修正候補の生成に着目する．既存のソースコードから API 利用パターンをマイニングしてデータベース化し，修正箇所周辺における API の利用方法と類似する API 利用パターンをデータベースから検索する．検索された API 利用パターンと修正箇所周辺のソースコードを比較し，修正対象のソースコードに不足しているメソッドの呼び出しを追加する修正候補を生成する．提案手法に基づいてプロトタイプツールを実装し，複数のソースコードについて修正できることを確認した．

1. はじめに

ソフトウェア開発におけるデバッグの負担を軽減するために，自動プログラム修正の技術が研究されている [1]．自動プログラム修正は，ソースコード中のソフトウェア障害（バグ）の原因である部分をコンピュータに修正させる技術である．自動プログラム修正は大きくわけて (1) 修正箇所の特定，(2) 修正候補の生成，(3) 修正候補の評価の 3 ステップで実行される．修正箇所の特定ではソースコード中の修正すべきバグの原因箇所を特定し，修正候補の生成では特定したバグの原因箇所に対する修正候補を複数生成する．修正候補の評価では生成された各修正候補が正しくバグを修正できるか判定する．自動プログラム修正の各ステップについて様々な手法が提案されている．自動プログラム修正では，バグを正しく修正できる修正候補を生成する必要があるが，ソースコードの変更は数限りなく可能である中で，バグを正しく修正できるものは一部のみである．このため，バグを正しく修正できる修正候補を生成するために，様々な修正候補生成のアプローチが研究されている．

代表的な修正候補の生成手法として，修正対象のプログラムなどに存在する既存のソースコード片を利用するものと，バグに対する修正方法を表す修正テンプレートに合わせてソースコードを変更するものがある．ソースコード片を利用する手法では，バグの修正に必要なソースコード片が存在していなければ，バグを修正できないという問題がある．修正テンプレートを利用する手法では，典型的なバグに対する修正方法を修正テンプレートとしてあらかじめ準備し，修正対象のバグに対して有効な修正テンプレートに沿ってソースコードを変更することでバグを修正する．このため，対象プログラムの内容に関係なく，定義した修正テンプレートが有効なバグを修正できる．しかし，様々なバグを修正するためには様々な修正テンプレートを用意する必要があり，修正テンプレートの作成には多大な労力を要するという問題がある．

ソフトウェア開発において様々な機能を提供する API の利用は必須である．API の中には，利用したい機能に対して呼び出すメソッドの種類と順序が決まっているなど，典型的な利用方法が存在する API も少なくない（以下，API の典型的な利用方法を API 利用パターンと呼ぶ）．このような API を利用する場合，API 利用パターンに従わない方法で API を利用するとバグの原因になる可能性がある．

本稿では，API 利用パターンをソースコード中に出現する順序で並べられた API 呼び出しの列と定義し，API 利用パターンを用いた自動プログラム修正手法を提案する．提案手法では，修正箇所の特定と修正候補の評価には既存手法を用いることとして，API 利用パターンに基づく修正候補の生成手法を構築する．既存のソースコードから API 利用パターンをマイニングしてデータベース化し，修正箇所周辺における API の利用方法と類似する API 利用パターン

[1] 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
[2] 南山大学 情報センター
Center for Information and Communication Technology, Nanzan University
[3] 立命館大学 情報理工学部
College of Information Science and Engineering, Ritsumeikan University

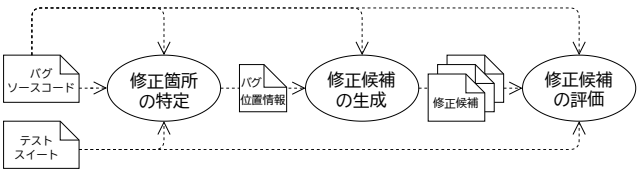

図 1 生成と検証に基づく自動プログラム修正の流れ

（以下，正解パターンと呼ぶ）をデータベースから検索する．検索された正解パターンと修正箇所周辺のソースコードを比較し，修正対象のソースコードに不足しているメソッドの呼び出しを修正対象のソースコードに追加するように修正候補を生成する．提案手法により，必要なメソッドを呼び出していないことが原因のバグを修正できる．

本稿の構成は以下の通りである．2 節で自動プログラム修正，3 節で API 利用パターンについて述べる．4 節で API 利用パターンを用いた自動プログラム修正を提案し，5 節でプロトタイプ実装について述べる．6 節で適用例を示し，7 節で関連研究，8 節まとめと今後の課題を述べる．

2. 自動プログラム修正

自動プログラム修正 [1] は，ソースコード中のソフトウェア障害（バグ）の原因箇所をコンピュータに修正させる技術である．図 1 のように，大きくわけて修正箇所の特定，修正候補の生成，修正候補の評価の 3 段階で実現される．

修正箇所の特定には自動バグ限局の技術が利用される．自動バグ限局は，失敗するテストケースからバグの原因箇所を特定する技術であり，様々な手法が提案されている [2], [3], [4]．例えば，SBFL(Spectrum Based Fault Localization)[4] では，成功したテストケースと失敗したテストケースそれぞれで実行された行の情報から，ソースコードの各行についてバグの可能性を表すスコアを算出する．スコアが高いほどバグを含む行である可能性が高い．

修正候補の生成では特定された修正箇所周辺を改変したソースコードを生成し，修正候補の評価では改変されたソースコードにおいてバグが修正されているか確認する．生成と検証に基づく手法とプログラムの意味論に基づく手法に大別できるが，前者では，修正候補の生成として何らかの戦略に基づいてソースコードを改変し，修正候補の評価として改変したソースコードがテストに成功するか確認する．この手法では，改変したプログラムが用意した全てのテストケースを正常に実行できるまで，修正対象のプログラムに対して繰り返し改変を行う．プログラムの意味論に基づく手法では，テストスイートから修正対象のプログラムが満たすべき条件を特定し，その条件を満たすようにソースコードを改変する．この場合，修正候補が生成できれば，そのソースコードはテストに成功する．

代表的な生成と検証に基づく手法として，遺伝的アルゴリズムを利用する手法 [5], [6] と修正テンプレートを利用する手法 [7], [8], [9] がある．遺伝的アルゴリズムを利用する手法は，修正対象のソースコードに対してソースコード片の挿入，削除，置換を繰り返すことで徐々にバグが修正された修正候補を得ようとする手法である．挿入や置換に用いるソースコード片を同一ファイル内，同一パッケージ内，同一プロジェクト内などから取得する手法が提案されている．修正テンプレートを利用する手法では，ソースコードを事前に準備された修正テンプレートに従って変更することでバグを修正する手法である．

Listing 1 API 利用パターンの例

```
1 java.util.List<java.lang.String>.stream,
2 java.util.stream.Stream<java.lang.String>.
      findFirst,
3 java.util.Optional<java.lang.String>.orElse
```

3. API 利用パターン

大規模ソフトウェアの開発コストを低減するために様々なフレームワークやライブラリが開発されており，フレームワークやライブラリが提供する機能を利用するためのアプリケーションプログラミングインターフェース (API) が定義されている．API は関数の呼び出しと同じように API 中で定義されているメソッドを呼び出すことで簡潔に利用できる．しかし，実際には API に対する制約を理解して正しく利用しないとバグの原因となる可能性がある．例えば，入出力 API では open メソッドの呼び出し 1 回に対して close メソッドの呼び出しが 1 回だけ存在するように利用することが規定されることがある．このような制約を守る正しい API の利用方法を API 利用パターン [10] と呼ぶ．

必要なメソッドを呼び出していない，不要なメソッドを呼び出している，メソッドを呼び出す順序が入れ替わっている，といった API 利用パターンに従わない方法で API を使用するとバグの原因となる可能性がある．例として，Listing1 の API 利用パターンと Listing1 に従わないソースコード Listing2 について考える．Listing1 は List クラスの stream メソッド，Stream クラスの findFirst メソッド，Optional クラスの orElse メソッドを順に呼び出すことを表す．Listing2 では List クラスの stream メソッド，Stream クラスの findFirst メソッド，Optional クラスの get メソッドを順に呼び出しているが，`args` が長さ 0 の配列の場合，findFirst メソッドが空の Optional を返すため get メソッド呼び出しで NoSuchElementException が発生するというバグが存在する．このバグは get メソッドの呼び出しが直接の原因であり，get メソッドを orElse メソッドに置換する，もしくは isPresent メソッドの返り値を条件とする if 文で get メソッドの呼び出しを囲むなどの方法で修正できる．

ここで，Listing1 の API 利用パターンに従うように Listing2 の get メソッドの呼び出しを orElse メソッドの呼び

Listing 2 API 利用パターンに従わないソースコード

```
1 public static void main(String[] args) {
2   List<String> a = Arrays.asList(args);
3   Stream<String> b = a.stream();
4   Optional<String> c = b.findFirst();
5   String d = c.get();
6 }
```

Listing 3 Listing2 に対する修正プログラム

```
1 public static void main(String[] args) {
2   List<String> a = Arrays.asList(args);
3   Stream<String> b = a.stream();
4   Optional<String> c = b.findFirst();
5   String d = c.orElse("");
6 }
```

出しに変更したものを Listing3 に示す．この変更により，findFirst メソッドの返り値が空の Optional であっても NoSuchElementException は発生しない．このように，API の利用方法が原因のバグはソースコードを API 利用パターンに合わせるように変更することで修正できる可能性がある．

API の利用方法が原因のバグを GenProg[5] 等の既存のソースコード片を活用する手法で修正する場合，修正に必要なメソッド呼び出しが既存のソースコード片として存在しなければバグを修正できない．Listing2 では，orElse メソッドの呼び出しが存在していればバグを修正できる可能性がある．修正テンプレートを利用する手法では，get メソッドを orElse メソッドに置換する修正テンプレートが存在すればバグを修正できる可能性がある．しかし，バグに対して有効な修正テンプレートはバグの原因となっている API に依存するため，API の利用方法が原因のバグに有効な修正テンプレートの作成は容易ではない．

4. API 利用パターンを用いた自動プログラム修正

4.1 概要

API 利用パターンを用いて API の誤った利用が原因のバグを自動で修正する手法を提案する．ここで，API 利用パターンを，呼び出される API メソッドの一意な名前をソースコード中に出現する順に並べた列と定義する．API の誤った利用には，必要なメソッドを呼び出していない，不要なメソッドを呼び出している，呼び出す順序が誤っている，などがある．提案手法では 1 つ目の誤用を対象として，必要なメソッド呼び出しをソースコードに追加するような修正を実現する．本研究ではオブジェクト指向言語（特に Java 言語）を対象とするが，提案手法はオブジェクト指向言語以外のプログラミング言語に対しても適用可能である．

提案手法の全体像を図 2 に示す．修正箇所の特定と修正候補の評価には既存手法を用いることとし，提案手法では API 利用パターンに基づく修正候補の生成手法を構築する．修正候補を生成するために，あらかじめ既存のソースコードからマイニングされた API 利用パターンが格納されたデータベース（以下，パターンデータベース）を準備する．修正候補の生成は，API 利用パターンの検索およびソースコード改変の 2 段階で実行される．API 利用パターンの検索では，特定された修正箇所周辺において呼び出される API メソッドの一意な名前の列（以下，バグパターン）に類似しているがわずかに異なる API 利用パターン（以下，正解パターン）をパターンデータベースから検索する．検索された正解パターンとバグパターンを比較して修正対象のソースコードに不足しているメソッドを特定し，特定されたメソッドを呼び出すように改変したソースコードを修正候補として生成する．

4.2 正解パターンの検索

ソースコードの修正方針となる正解パターンをパターンデータベースから検索する．パターンデータベース中の API 利用パターン（以下，データベースパターン）とバグパターンの類似度を算出し，類似度が高いデータベースパターンを正解パターンとする．ここで，データペースパターン P とバグパターン B の類似度を，P の部分列 P' のうち以下のように定義される条件 $S(P';B)$ を満たす P' の中で $|P'|$ の最大値を n として $n/|P|$ と定義する．$|P|$ は列 P の要素数である．

- $S(p;B) = p \in B$
- $S(p_1,\ldots,p_m;b_1,\ldots,b_n) =$
  $\exists j.p_1 = b_j \wedge S(p_2,\ldots,p_m;b_{j+1},\ldots,b_n)$

直観的には，データベースパターンに含まれるメソッドのうちバグパターン中に順序を保ったまま出現するメソッドの割合の最大値である．

例えば，バグパターン

```
java.util.Arrays.asList,
java.util.List.add,
java.util.List.stream,
java.util.stream.Stream.filter,
java.util.stream.Stream.map,
java.util.stream.Stream.findFirst,
java.util.Optional.get
```

に対し，データベースパターン

```
java.util.List.stream,
java.util.stream.Stream.map,
java.util.stream.Stream.filter,
java.util.stream.Stream.findFirst,
java.util.Optional.orElse
```

を考える．このとき，データベースパターンに含まれるメソッド呼び出しのうちバグパターン中に順序を保ったまま出現するメソッド呼び出しの列は

```
java.util.List.stream,
```

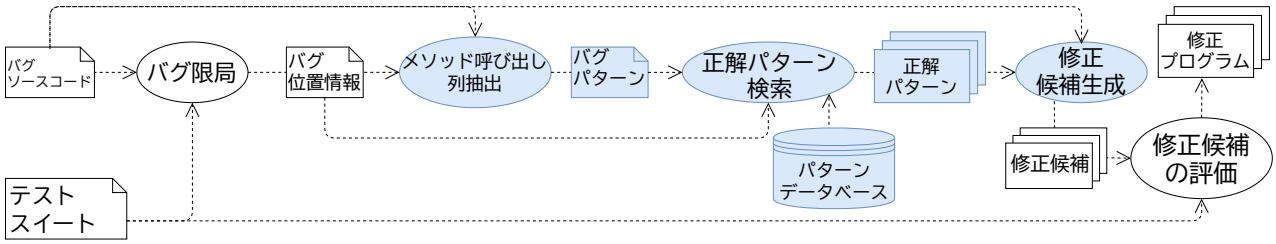

図 2 API 利用パターンを用いた自動プログラム修正

`java.util.stream.Stream.map`,
`java.util.stream.Stream.findFirst`
と
`java.util.List.stream`,
`java.util.stream.Stream.filter`,
`java.util.stream.Stream.findFirst`
の 2 つである．2 つとも長さが 3，データベースパターンは長さが 5 のため，類似度は $(3 \div 5) \times 100 = 60\%$ である．

加えて，類似度が同一のデータベースパターンのうちバグパターン中に順序を保ったまま出現するメソッド呼び出しの数が多いものをより正解パターンとして適しているとみなす．ただし，類似度 100%のデータベースパターンはバグパターンに完全一致であり修正方針とはならないため，正解パターンとはみなさない．

4.3 修正候補の生成

正解パターンを利用して不足しているメソッド呼び出しを追加する修正候補を生成する．具体的には，修正対象のソースコード上で追加すべきメソッドの呼び出しを文として挿入できる位置を列挙し，各挿入位置について引数とレシーバーの候補を型に基づいて列挙する．レシーバーの候補と引数の候補を組み合わせて追加するメソッド呼び出し式を生成し，生成したメソッド呼び出し式の返り値を新しい変数に格納する変数宣言文を挿入する．最後に，挿入した変数宣言文の変数と型が一致する箇所を変数宣言文の変数で置き換えたプログラムを修正候補として出力する．

4.3.1 不足しているメソッドの特定

正解パターンとバグパターンを比較して修正対象のソースコードに不足しているメソッドを特定する．正解パターンに含まれるメソッドの中でバグパターン中に順序を保ったまま出現するメソッドの列を列挙し，列挙された中で最も長い列に出現しない正解パターン中のメソッドを不足しているメソッドとする．

例えば，バグパターン
`java.util.Arrays.asList`,
`java.util.List.add`,
`java.util.List.remove`,
`java.util.List.indexOf`,
`java.util.List.stream`,
`java.util.stream.Stream.findFirst`,
`java.util.Optional.get`
に対し正解パターンを
`java.util.List.stream`,
`java.util.stream.Stream.findFirst`,
`java.util.Optional.orElse`
とすると，正解パターンに含まれるメソッドのうちバグパターン中に順序を保ったまま出現するメソッドの列は
`java.util.List.stream`
と
`java.util.stream.Stream.findFirst`
と
`java.util.List.stream`,
`java.util.stream.Stream.findFirst`
の 3 通りである．この中で最長の列は 3 つ目の列であり，3 つ目の列に含まれず正解パターンに含まれるメソッドは `java.util.Optional.orElse` である．このことから，不足しているメソッドを java.util.Optional クラスの orElse メソッドであると特定する．

4.3.2 挿入位置の決定

不足しているメソッドを，正解パターンにおける不足メソッド前後のメソッドの呼び出しの間に挿入する．メソッド呼び出しを式として挿入する場合，挿入可能な位置が多岐にわたる上，不足メソッド前後のメソッドがあるメソッド呼び出しの第一引数と第二引数に出現しているなど単純には挿入できない場合がある．そこで本手法では，不足メソッドの呼び出しによる初期化付きの変数宣言文として不足メソッドを挿入する．メソッド呼び出しの挿入を文の挿入として実現するため，1 つの式文が高々 1 つのメソッド呼び出し式を含むようにソースコードを変形する．これにより，不足メソッド前後のメソッドそれぞれが呼び出される 2 つの文の間を挿入位置の候補とできる．なお，宣言した変数は不足メソッドの返り値が必要な位置で利用する．

例えば，Listing4 は Listing5 のように変形される．Listing4 の 2 行目の引数のメソッド呼び出しは Listing5 の 2 行目と 3 行目，Listing4 の 3 行目のメソッドチェーンは Listing5 の 4 行目から 6 行目のように分解される．もし stream と findFirst の間に limit が不足しているのであれば，Listing5 の 4 行目と 5 行目の間が挿入位置となる．

Listing 4 変形前のソースコード例

```
1 List<String> a = Arrays.asList(args);
2 a.remove(a.size() - 1);
3 String d = a.stream().findFirst().get();
```

Listing 5 Listing4 を変形したソースコード

```
1 List<String> a = Arrays.asList(args);
2 int size = a.size();
3 a.remove(size - 1);
4 Stream<String> b = a.stream()
5 Optional<String> c = b.findFirst()
6 String d = c.get();
```

4.3.3 引数とレシーバーの候補の決定

挿入位置の候補それぞれについて挿入するメソッド呼び出しの引数とレシーバーの候補集合を生成する．不足メソッドの引数とレシーバーの型に基づいて，主に変数とリテラルを候補として選択する．挿入位置で利用可能な変数は，フィールド，仮引数，挿入位置以前で宣言されているローカル変数である．さらに，挿入位置で利用可能なコレクションや配列の要素も型が一致すれば候補とできる．型がプリミティブ型もしくはプリミティブ型のラッパークラスであれば，そのプリミティブ型のデフォルト値のリテラルを候補に含める．String 型であれば変数以外に空文字列も候補に追加する．

4.3.4 挿入する変数宣言文の生成

引数の候補とレシーバーの候補を組み合わせて挿入するメソッド呼び出し式を生成し，メソッド呼び出しの返り値によって初期化される新しい変数の宣言文を生成する．

例として，Listing6 に対して正解パターンが

`java.util.List.stream`,
`java.util.stream.Stream.findFirst`,
`java.util.Optional.orElse`

であり，orElse メソッドが不足している場合を考える．このとき，正解パターンにおいて orElse メソッドの直前が findFirst メソッドであるため，挿入位置の候補は 5 行目と 6 行目の間，6 行目と 7 行目の間，7 行目と 8 行目の間である．orElse メソッドのレシーバーの型は Optional クラスであるため，各挿入位置について挿入位置以前で宣言された Optional クラスの変数をレシーバーの候補とする．orElse メソッドの引数の型はジェネリックであるが Listing6 では String であるため，各挿入位置について挿入位置以前に宣言された String 型の変数を引数の候補とする．5 行目と 6 行目の間に挿入する場合，レシーバーには 5 行目の変数 c のみ，引数には 4 行目の変数 b のみ指定できるため，

```
String g = c.orElse(b);
```

を挿入する．6 行目と 7 行目の間に挿入する場合，レシーバーには 5 行目の変数 c と 6 行目の変数 f，引数には 4 行目の変数 b のみ指定できるため，

Listing 6 メソッド呼び出し挿入の例題プログラム

```
1 public static void main(String[] args) {
2   List<String> a = Arrays.asList(args);
3   Stream<String> e = a.stream();
4   String b = "NULL";
5   Optional<String> c = b.findFirst();
6   Optional<String> f = Optional.empty();
7   String d = c.get();
8 }
```

```
String g = c.orElse(b);
String g = f.orElse(b);
```

の 2 つが挿入される変数宣言文の候補となる．7 行目と 8 行目に間に追加する場合も同様である．

4.3.5 返り値の利用箇所の決定

バグの修正には不足しているメソッド呼び出しを挿入するだけではなく，挿入したメソッド呼び出しの返り値の適切な利用が必要な場合がある．そのため，不足メソッドの呼び出し式を含む変数宣言文の挿入に加え，挿入位置以降に出現する式のうち返り値の型と一致する式を宣言した変数で置換する．本手法では高々 1 つの式を置換対象とする．同時に複数の式を置換することも可能であるが，その場合は置換可能な式の数に対して指数的に多くの修正候補が生成される．変数宣言文の挿入と，宣言した変数で一部の式を置換したソースコードを修正候補とする．

例えば，Listing6 の 5 行目と 6 行目の間に

```
String g = c.orElse(b);
```

を挿入する場合，返り値の利用箇所の候補は 7 行目の右辺のみであるため 7 行目の右辺を g に置換できる．同様に，6 行目と 7 行目に

```
String g = c.orElse(b);
String g = f.orElse(b);
```

のいずれかを挿入する場合でも 7 行目の右辺を g に置換できる．挿入位置が 7 行目と 8 行目の間の場合も同様である．

5. 実装

提案手法をプロトタイプツールとして Java 言語で実装する．修正対象を Java プログラムとする．修正箇所の特定と修正候補の評価は既存手法に基づいて手動で実行する．修正箇所として特定された行を含むメソッドを修正対象とし，対象メソッド内で呼び出されるメソッドの列をバグパターンとする．

5.1 パターンデータベースの生成

既存のソースコードをマイニングして得られた API 利用パターンを格納したデータベースを生成する．マイニングツールとして Java プログラムから API 利用パターンをマイニングする PAM(Probabilistic API Mining)[11] を用いる．PAM はメソッドの完全修飾名のリストとして API 利用パ

Listing 7 被呼び出しメソッドのリスト生成のプログラム例

```
1 public static void main(String[] args) {
2   List<String> a = Arrays.asList(args);
3   Stream<String> b = a.stream();
4   Optional<String> c = b.findFirst();
5   String d = c.orElse("");
6 }
```

ターンを生成する．PAM の入力は，メソッド名とそのメソッド内で呼び出されるメソッドの完全修飾名のリストを組とする ARFF ファイルである．今回の実装では，Java プログラムのバイトコードを静的解析するためのクラスライブラリである SOBA[12] を利用して ARFF ファイルを生成する．SOBA を用いてクラスファイルを解析し，マイニング対象の各メソッドの内部で呼び出されるメソッドの完全修飾名をバイトコードに出現する順序で並べる．

例えば，Listing7 の main メソッドから生成されるメソッドの完全修飾名のリストは

```
java.util.Arrays.asList
java.util.List.stream
java.util.stream.Stream.findFirst
java.util.Optional.get
```

である．

5.2 正解パターンの検索

バグパターンはバグ限局で特定された文を含むメソッドから ARFF ファイル生成時と同様の方法で生成する．すべてのデータベースパターンについてバグパターンとの類似度を算出し，類似度が上位 2 位までのデータベースパターンを正解パターンとする．

5.3 修正候補の生成

修正対象のソースコードの変更は Java プログラムの抽象構文木を操作できるライブラリである JavaParser[*1]を用いて実現する．修正対象のソースコードに対してメソッドチェーンや実引数に指定されるメソッド呼び出しを分解する変形は未実装である．提案手法ではフィールドを引数やレシーバーの候補に含めるようにしているが，現在の実装ではフィールドは扱えない．JavaParser は静的解析ツールであるためコレクションの要素の具体的な型を取得できない場合があり，現在の実装ではコレクションの要素を引数やレシーバーの候補として利用していない．

6. 適用例

6.1 パターンデータベース

適用例のために，Java 言語で実装されている 156 個のプロジェクトに含まれる 34716 個のメソッドを対象に，java.lang パッケージと java.util パッケージのメソッドの呼び出しのみを抽出してパターンデータベースを生成した．生成されたパターンデータベースには 4302 個の API 利用パターンが含まれている．

[*1] https://javaparser.org/

Listing 8 修正対象のプログラム

```
1  public String example(String[] args) {
2    Stream<String> stream = Arrays.stream(args);
3    double tx = ...;
4    double ty = ...;
5    Stream<String> s = stream
6      .filter(t -> {
7        double x = getX(t);
8        double y = getY(t);
9        return (x == tx && y == ty);
10     });
11   Optional<String> a = s.findFirst();
12   String result = a.get();
13   return result;
14 }
```

6.2 StackOverFlow

StackOverFlow で公開されているプログラム [13] のバグを提案手法によって修正する例を示す．対象プログラムを Listing8 に示す．元のプログラムでは findFirst や get の呼び出しはメソッドチェーンとして記述されているが，提案手法を適用するために変形されている．11 行目の findFirst の結果が空の Optional であれば，11 行目の get メソッドの呼び出しで NoSuchElementException が発生する．

6.2.1 正解パターンの検索

Listing8 に対する正解パターンをパターンデータベースから検索する．Listing8 のバグパターンは

```
java.util.Arrays.stream,
java.util.stream.Stream.filter,
java.util.stream.Stream.findFirst,
java.util.Optional.get
```

であり，このバグパターンと各データベースパターンの類似度を計算する．100%を除き類似度が上位 2 位までのデータベースパターンを Listing9 に示す．これらを正解パターンとして，バグパターン中に順序を保って出現するメソッド呼び出しの数の多いパターンから修正候補を生成する．

6.2.2 修正候補の生成

この例では，類似度 66%のデータベースパターン

```
java.util.stream.Stream.filter,
java.util.stream.Stream.findFirst,
java.util.Optional.orElse
```

を正解パターンとして修正候補を生成する．この正解パターンとバグパターンを比較すると orElse メソッドの不足がわかるため，orElse メソッドの呼び出しを追加するようにソースコードを修正する．メソッド呼び出しの挿入位置の候補集合を生成し，各挿入位置候補についてレシーバー，引数，返り値の利用箇所の候補集合を生成する．

Listing 9 Listing8 に対する正解パターンと類似度

```
1  ---------- 類似度 66%
2  [java.util.List.stream java.util.stream.Stream.filter java.util.stream.Stream.findFirst ]
3  [java.util.stream.Stream.filter java.util.stream.Stream.findFirst java.util.Optional.orElse ]
4  [java.util.Arrays.stream java.util.stream.Stream.filter java.util.stream.Stream.map ]
5  [java.util.Arrays.stream java.util.stream.Stream.map java.util.stream.Stream.filter ]
6  ---------- 類似度 50%
7  [java.util.Optional.isPresent java.util.Optional.get ]
8  [java.util.stream.Stream.filter java.util.stream.Stream.collect ]
9  [java.util.stream.Stream.filter java.util.stream.Stream.forEach ]
10 [java.util.Arrays.stream java.util.stream.Stream.anyMatch ]
11 [java.util.Arrays.stream java.util.stream.Stream.forEach ]
12 [java.util.List.stream java.util.stream.Stream.filter ]
13 [java.util.Set.stream java.util.stream.Stream.filter ]
14 [java.util.Arrays.stream java.util.stream.Stream.map ]
```

正解パターンにおいて orElse メソッドの直前は findFirst メソッドであり，Listing8 で findFirst メソッドは 11 行目で呼び出されている．さらに，正解パターンにおいて orElse メソッドは末尾である．以上より，orElse メソッドの挿入位置の候補は 11 行目と 12 行目の間，および 12 行目と 13 行目の間である．

orElse メソッドのレシーバーの型は Optional クラスであり，11 行目と 12 行目の間に挿入する場合に利用可能な Optional クラスの変数は 11 行目で宣言される変数 a のみである．12 行目と 13 行目に挿入する場合も同様に変数 a のみを利用できる．そのため，いずれに挿入する場合も 11 行目の変数 a をレシーバーとする．orElse メソッドの引数の型はジェネリックであるが Listing8 では String クラスである．11 行目と 12 行目の間に挿入する場合に利用可能な String クラスの値は仮引数の配列 args の要素のみであるが，現在の実装ではコレクションと配列を扱えないため，引数にはリテラルの空文字列を指定する．12 行目と 13 行目に挿入する場合も同様である．

引数とレシーバーの候補から，挿入する変数宣言文を生成する．挿入位置候補のいずれに挿入する場合もレシーバーの候補は 11 行目の変数 a，引数の候補は空文字列であるため，生成するメソッド呼び出し式は 1 通りである．orElse メソッドの返り値は String クラスであり，Listing8 において利用可能な新しい変数名として b を利用すると，挿入する変数宣言文は

```
String b = a.orElse("");
```

となる．

最後に，挿入位置以降で変数 b で置換可能な箇所を特定する．挿入位置が 11 行目と 12 行目の間の場合，置換可能な箇所は 12 行目の右辺と 13 行目の return 文の値式である．挿入位置が 12 行目と 13 行目の間の場合は 13 行目の return 文の値式のみである．そのため，修正プログラムは Listing10 から Listing14 の 5 通り生成される．いずれも 11 行目以降のみ示す．Listing10 から Listing12 では 12 行目に orElse の呼び出しが挿入されている．Listing10 では置換なし，Listing11 では代入の右辺，Listing12 では return 文の値式が置換されている．Listing13 と Listing14 では 13 行目に orElse の呼び出しが挿入されている．Listing13 では置換なし，Listing14 では return 文の値式が置換されている．

Listing 10 修正候補その 1

```
11   Optional<String> a = s.findFirst();
12   String b = a.orElse("");
13   String result = a.get();
14   return result;
15 }
```

Listing 11 修正候補その 2

```
11   Optional<String> a = s.findFirst();
12   String b = a.orElse("");
13   String result = b;
14   return result;
15 }
```

Listing 12 修正候補その 3

```
11   Optional<String> a = s.findFirst();
12   String b = a.orElse("");
13   String result = a.get();
14   return b;
15 }
```

Listing 13 修正候補その 4

```
11   Optional<String> a = s.findFirst();
12   String result = a.get();
13   String b = a.orElse("");
14   return result;
15 }
```

Listing 14 修正候補その 5

```
11   Optional<String> a = s.findFirst();
12   String result = a.get();
13   String b = a.orElse("");
14   return b;
15 }
```

Listing11 では get メソッドの呼び出しが削除されており，レシーバーが空の Optional であっても NoSuchElementException は発生しない．これに対して Listing10 と Listing12 では 12 行目に orElse の呼び出しが挿入されているが，13 行目で get メソッドが呼び出されるため NoSuchElementException が発生する可能性がある．Listing13 と Listing14 についても 12 行目の get メソッドの呼び出しで NoSuchElementException が発生する可能性がある．そのため Listing11 のみが Listing8 の修正プログラムになっている．

7. 関連研究

Nielebock は修正対象のソースコードに対して有効な API 利用パターンをマイニングし，取得した API 利用パターンを用いてバグを自動修正するシステムを提案している [14].

再利用を目的として定義された API を正しく利用するために API 利用パターンが活用されている．Alur らは Java 言語の API について有限状態パターンを推論する手法 JIST を提案している [15]．Michail は関連性探索手法を用いて API 利用パターンをマイニングする手法を提案している [16], [17]．Michail の手法は開発者にプログラムの再利用方法を示す CodeWeb で利用されている [18].

8. おわりに

本稿では，API 利用パターンをソースコード中に出現する順序で並べられた API 呼び出しの列と定義し，API 利用パターンを用いて自動プログラム修正における修正候補の生成を行う手法を提案した．既存のソースコードから API 利用パターンをマイニングしてデータベース化し，修正箇所周辺における API の利用方法と類似する API 利用パターンをデータベースから検索する．検索された API 利用パターンの中で呼び出されていないメソッドを特定し，その呼び出しを追加するような修正候補を生成する．提案手法をプロトタイプツールとして実装し，例題プログラムを修正できることを確認した．

今後の課題として，不要なメソッド呼び出しの削除や呼び出す順序の入れ替えへの対応が挙げられる．提案手法では，修正対象のソースコードに対し開発者が意図しない変形を行うため，生成される修正候補の理解が難しくなる可能性がある．変形を可能な限り元に戻すことは今後の課題である．提案手法を GenProg などの既存手法と組み合わせることも今後の課題である．

謝辞 本研究の一部は JSPS 科研費 JP17K12666, JP18K11241 および 2020 年度南山大学パッヘ研究奨励金 I-A-2 の助成による．

参考文献

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

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

[3] Chen, M. Y., Kiciman, E., Fratkin, E., Fox, A. and Brewer, E.: Pinpoint: problem determination in large, dynamic Internet services, *Proceedings International Conference on Dependable Systems and Networks*, pp. 595–604 (2002).

[4] Harrold, M. J., Rothermel, G., Sayre, K., Wu, R. and Yi, L.: An empirical investigation of the relationship between spectra differences and regression faults, *Software Testing, Verification and Reliability*, Vol. 10, No. 3, pp. 171–194 (2000).

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

[6] Martinez, M. and Monperrus, M.: ASTOR: A Program Repair Library for Java, *Proceedings of ISSTA, Demonstration Track*, pp. 441–444 (2016).

[7] Pei, Y., Furia, C. A., Nordio, M., Wei, Y., Meyer, B. and Zeller, A.: Automated Fixing of Programs with Contracts, *IEEE Transactions on Software Engineering*, Vol. 40, No. 5, pp. 427–449 (2014).

[8] Pei, Y., Wei, Y., Furia, C. A., Nordio, M. and Meyer, B.: Code-based automated program fixing, *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pp. 392–395 (2011).

[9] Long, F. and Rinard, M.: Automatic Patch Generation by Learning Correct Code, *SIGPLAN Not.*, Vol. 51, No. 1, pp. 298–312 (2016).

[10] Robillard, M. P., Bodden, E., Kawrykow, D., Mezini, M. and Ratchford, T.: Automated API Property Inference Techniques, *IEEE Transactions on Software Engineering*, Vol. 39, No. 5, pp. 613– –637 (2013).

[11] Fowkes, J. and Sutton, C.: Parameter-free probabilistic API mining across GitHub, *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2016).

[12] 秦野智臣，石尾　隆，井上克郎：SOBA:シンプルな Java バイトコード解析ツールキット，コンピュータソフトウェア，Vol. 33, No. 4, pp. 4_4–4_15 (2016).

[13] stackoverflow.com: Avoid NoSuchElementException with Stream, `https://stackoverflow.com/questions/30686215/avoid-nosuchelementexception-with-stream`.

[14] Nielebock, S.: Towards API-specific automatic program repair, *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1010–1013 (2017).

[15] Alur, R., Černý, P., Madhusudan, P. and Nam, W.: Synthesis of Interface Specifications for Java Classes, *SIGPLAN Not.*, Vol. 40, No. 1, pp. 98–109 (2005).

[16] Michail, A.: Data mining library reuse patterns in user-selected applications, *14th IEEE International Conference on Automated Software Engineering*, pp. 24–33 (1999).

[17] Michail, A.: Data mining library reuse patterns using generalized association rules, *Proceedings of the 2000 International Conference on Software Engineering. ICSE 2000 the New Millennium*, pp. 167–176 (2000).

[18] Michail, A.: Code Web: Data Mining Library Reuse Patterns, *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, IEEE Computer Society, pp. 827–828 (2001).