社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

シグネチャ情報と入出力情報を用いたJavaメソッドの生成

下仲　健斗† 　肥後　芳樹† 　松本淳之介† 　内藤　圭吾† 　楠本　真二†

† 大阪大学大学院情報科学研究科，吹田市

E-mail: †{s-kento,higo,j-matumt,k-naitou,kusumoto}@ist.osaka-u.ac.jp

あらまし　ソースコードを自動的に生成する，自動プログラミングと呼ばれる技術は古くから研究されている．これまでいくつかの手法が提案されてきているが，用途が限定的であるものや，数行程度の短いソースコードしか生成できないものなどがほとんどである．本研究では，生成の対象を Java メソッドに限定し，Java メソッドの仕様からソースコードを自動生成することを試みる．仕様は，シグネチャ情報（引数の型と返値の型，メソッド名）および入出力情報（引数の値と返値の組の集合）である．提案手法では，シグネチャ情報を用いて既存の Java ソースコードを探索し，生成する Java メソッドの基となりうるコードを発見する．そして，入出力情報を満たすように少しずつコードを加工する．提案手法を評価するために，4 つのオープンソースプロジェクトに対して本手法を適用した．その結果，18 個の Java メソッドを自動生成することに成功した．生成された Java メソッドの事例を紹介し，手法の改善策を述べる．

キーワード　自動プログラミング，自動プログラム修正

1. は じ め に

自動プログラミングとは，開発者が作成したいソースコードを自動的に生成する技術のことである．つまり，開発者が直接ソースコードを書くのではなく，欲しいソースコードの挙動や仕様を断片的に与えると，それを満たすソースコードを出力する技術である．自動プログラミングという技術は古くから研究されており，これまでいくつかの手法が提案されてきている．

入力として，プログラムの概形を与え，細部を自動的に埋めてくれるスケッチングという手法が提案されている [1]．プログラムの概形とは，コードの一部 (定数など) が欠けている状態のプログラムである．開発者は，コードを完全に書く必要がなく，書かれていない部分が自動的に補われる．スケッチングを用いる場合は，開発者はプログラムの概形と同時にテストケースも与える必要がある．スケッチングは，そのテストケースを満たすように SAT ソルバを用いて細部を補う．

また，自然言語を入力とする自動プログラミング手法も提案されている．Gvero らは，関数呼び出しの自動補完ツールである AnyCode を開発した [2]．AnyCode では，関数名を入力するのではなく，呼び出したい関数の処理内容を自然言語で入力する．入力された自然言語を AnyCode が解析し，ユーザの意図に適した関数の候補が提示される．

Galuwani らは，文字列操作の自動化ツールである FlashFill(注1) を開発した [3]．FlashFill は Excel に実装されている機能であり，ユーザが与えた入出力例から，ユーザの意図を推量し，自動で文字列操作を行う．

このように，自動プログラミングの手法はこれまでいくつか提案されてきているが，次の 3 つの理由から実用化が困難といわれている [4]．

- ユーザの意図を入力として表現する難しさ
- 候補となるプログラムを抽出する難しさ
- 大規模なプログラム作成の難しさ

本研究では，生成の対象を Java メソッドに限定し，Java メソッドの仕様からソースコードを自動生成することを試みる．入力として与える Java メソッドの仕様は以下の 2 種類とする．

- シグネチャ情報 (引数の型，返値の型，およびメソッド名)
- 入出力情報 (引数の値と返値の組の集合)

ソフトウェア開発のプロセスでは，ソフトウェアの要件定義および設計に始まり，コーディングとテストを行い，運用と保守のフェーズに入る．このプロセスはウォーターフォールと呼ばれ，ソフトウェア工学における最も有名な開発モデルである．本手法における入力は，ソフトウェアの設計により定まった仕様であり，提案手法はその後のコーディングフェーズを自動化する手法といえる．提案手法の流れとしては，シグネチャ情報を用いて既存の Java ソースコードを探索し，生成する Java メソッドの基となりうるコードを発見する．そして，入出力情報 (以降テストケースと呼ぶ) を満たすように少しずつコードを加工する．提案手法が入力として必要とするシグネチャ情報はソフトウェアの設計フェーズで作成される情報であり，入出力情報は単体テストのフェーズで作成される情報である．つまり，提案手法を用いるために開発者は追加の作業を行う必要はない．

提案手法を評価するために，4 つのオープンソースプロジェ

(注1)：https://goo.gl/PNLmfc

```
1 public boolean startsWith(final String str){
2   if (str == null) {
3       return false;
4   }
5   final int len = str.length();
6   if (len == 0) {
7       return true;
8   }
9   if (len > (size)) {
10      return false;
11  }
12  for (int i = 0; i < len; i++) {
13      if ((buffer[i]) != (str.charAt(i))) {
14          return false;
15      }
16  }
17  return true;
18 }
```

(a) startsWith メソッド

```
1 public boolean endsWith(final String str){
2  if (str == null){
3    return false;
4  }
5  final int len = str.length();
6  if (len == 0) {
7    return true;
8  }
9  if (len > (size)){
10   return false;
11 }
12  int pos = size - len;
13  for (int i = 0; i < len; i++ , pos++){
14      if ((buffer[pos]) != (str.charAt(i))){
15          return false;
16      }
17  }
18  return true;
19 }
```

(b) endsWith メソッド

図 1: startsWith メソッドと endsWith メソッド

クトに対して実験を行った．実験では，プロジェクトからあるメソッドを除外した際に，残りのメソッドを加工することで，除外したメソッドの自動生成を試みた．この処理を合計 1,244 のメソッドに対して行った．実験の結果，18 個の Java メソッドを自動生成することに成功した．

以降，2. では研究の動機について述べ，3. で提案手法について説明する．4. で実装の準備として自動プログラム修正ツール GenProg について述べる．5. では提案手法の実装について説明する．6. では，適用実験について述べ，7. では実験に対する考察を述べる．最後に 8. で本研究のまとめと今後の予定について述べる．

2. 研究の動機

オープンソースプロジェクトである apache-commons-text(以下 commons-text) 内に，処理内容が類似した 2 つのメソッド，startsWith メソッドと endsWith メソッドが存在する．startsWith メソッドは，ある文字列 (文字列 1 とする) が入力となる文字列 (文字列 2 とする) で始まるかどうか判定するメソッドであり，endsWith メソッドは文字列 1 が文字列 2 で終わるかどうか判定するメソッドである．図 1 に startsWith メソッドと endsWith メソッドのソースコードを示す．図 1 から，処理内容だけでなく，ソースコード自体も似通ったものであることがわかる．

ここで，startsWith メソッドが commons-text 内に存在していない状況を想定する．開発者は，文字列 1 が文字列 2 で始まるかどうか判定するようなメソッドを作成したい．もし開発者が，endsWith メソッドがプロジェクト内に既に存在していることを認識しているならば，コピーアンドペーストを行い，既存コードを少し書き換えることで startsWith メソッドを実装することができる．しかし，endsWith メソッドの存在を開発者が認識していない場合，上記の方法をとることはできない．

そこで本研究では，メソッドの仕様を入力として与えることで，目的のメソッドを自動生成する手法を提案する．上述した状況において，startsWith メソッドを生成したい場合，提案手法の入力におけるシグネチャ情報を以下のように与える．

```
1 public boolean startsWith(final String str){
2  if (str == null){
3    return false;
4  }
5  final int len = str.length();
6  if (len == 0) {
7    return true;
8  }
9  if (len > (size)){
10   return false;
11 }
12  -int pos = size - len;
12  +int pos = 0;
13  for (int i = 0; i < len; i++ , pos++){
14   if ((buffer[pos]) != (str.charAt(i))){
15     return false;
16   }
17 }
18 return true;
19  }
```

図 2: 生成された startsWith メソッド

- メソッド名：startsWith
- 引数の型：String
- 返値の型：boolean

また，入出力情報を以下のように与える．

入出力例 1　文字列 1：abcd，文字列 2：ab，出力：true

入出力例 2　文字列 1：abcd，文字列 2：abcd，出力：true

入出力例 3　文字列 1：abcd，文字列 2：cd，出力：false

図 2 に，提案手法を用いて得られた startsWith メソッドを示す．生成されたコードでは，12 行目で文の置換が行われていることがわかる．このように，処理内容が類似したメソッドを加工することによって，自動生成が可能であることがわかる．

3. 提 案 手 法

本研究では，Java メソッドの仕様からソースコードを自動生成する手法を提案する．提案手法の概要を図 3 に示す．提案手法の入力は Java メソッドのシグネチャ情報，およびテストケースである．出力は，仕様を満たす Java メソッドである．

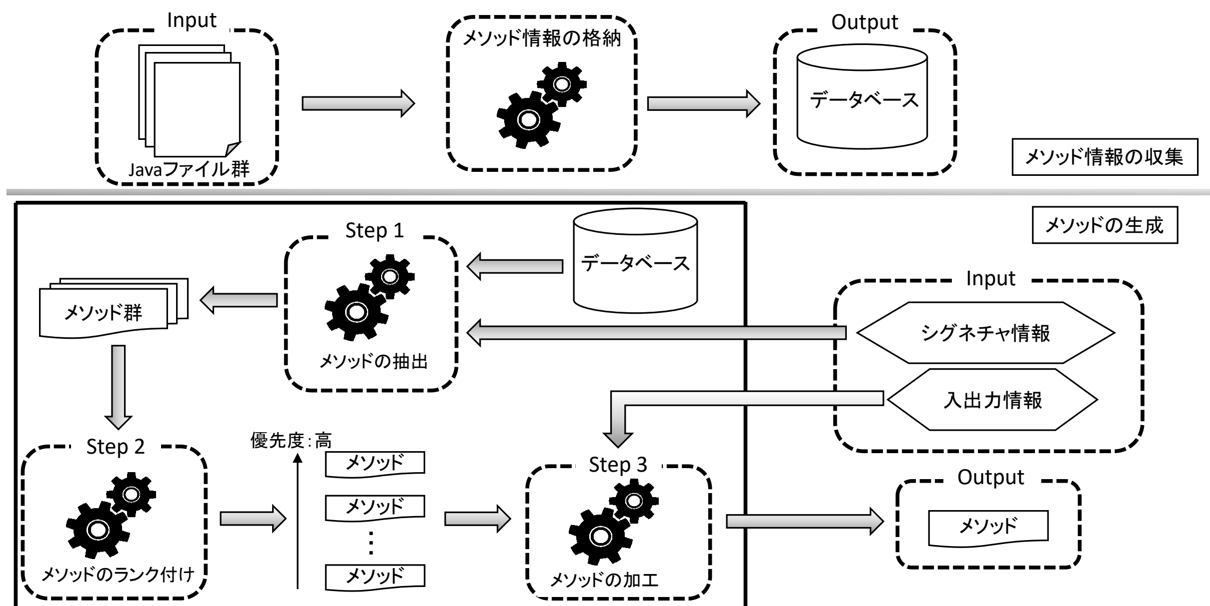

図 3: 提案手法の概要

提案手法は以下の 2 つの工程から構成される．

- メソッド情報の収集
- メソッドの生成

メソッド情報の収集では，プロジェクト内の Java ファイル群を解析し，各メソッドのシグネチャ情報をデータベースに格納する．

メソッドの生成工程は，次の 3 ステップから構成される．

Step 1　メソッドの抽出
Step 2　メソッドのランク付け
Step 3　メソッドの加工

Step 1 では，入力されたシグネチャにおける引数と返値の型と一致するメソッドを，あらかじめ用意しておいたデータベースから抽出する．そして Step 2 では，抽出したメソッド群を優先度が高い順に並び替える．これは，抽出したメソッドの数が多い場合に，ランダムに選択すると自動生成に必要な時間が膨大になる可能性があるからである．本手法では，入力されたシグネチャにおけるメソッド名との類似度を優先度の基準とする．Step 3 では，テストケースを満たすまで抽出したメソッドを加工する．

4. GenProg

本章では，実装で使用するツール，GenProg について述べる．GenProg は Weimer らが開発した自動プログラム修正ツールである [5]．自動プログラム修正ツールが担う主な機能は，欠陥箇所の限局と修正である．欠陥箇所の限局の手法として GenProg は，テストケース毎の実行パスから算出した疑惑値に基づいて順序付けを行う手法を用いている [6]．また，欠陥箇所の修正の手法として GenProg は，自動プログラム修正の手法の 1 つである再利用に基づく手法を採用している．この手法では，修正対象プログラムに存在するプログラム文を用いて欠陥箇所を変更したプログラムを生成する．GenProg は遺伝的プログラミングに基づき，この操作を変更したプログラムが全てのテストケースを通過するまで繰り返し行う．プログラムの変更は，プログラム文単位で行われ，具体的な処理は以下の 3 つである．

挿入　欠陥箇所にプログラム文の挿入を行う処理
削除　欠陥箇所を削除する処理
置換　削除処理と挿入処理を同時に行う処理

既存研究において，GenProg は 8 つのオープンソースソフトウェアに対して適用され，105 個中 55 個の欠陥の修正に成功している [7]．しかし，GenProg は既存のプログラム文を用いて変更を行うため，プログラム中に存在しない文が必要な欠陥は修正できないことや [8]，修正に要する時間が膨大であるといった課題もある [7]．

GenProg は元々OCaml で開発されているが，Java で再実装された jGenProg(注2) が公開されている．Martinez らは，3 つの自動プログラム修正ツール (GenProg, Kali [9], MutRepair [10]) を Java で再実装し，Astor というツールにまとめている [11]．本研究では Astor に内包されている jGenProg を用いる．

5. 実　　装

提案手法の実装について述べる．ただし，対象とするプロジェクトは以下の条件を満たすものである．

- Java で開発されている
- JUnit を用いたテストケースが存在する

提案手法の実装は，以下の 3 つの要素からなる．

- メソッド情報を格納するデータベース
- メソッドのランク付け
- メソッドの加工

以降，3 つの実装方法について述べる．

5.1 データベース

データベースとして SQLite [12] を用いた．データベースに格納する属性は，引数と返値の型，メソッド名，ファイルパス，

(注2)：https://goo.gl/9HzZbE

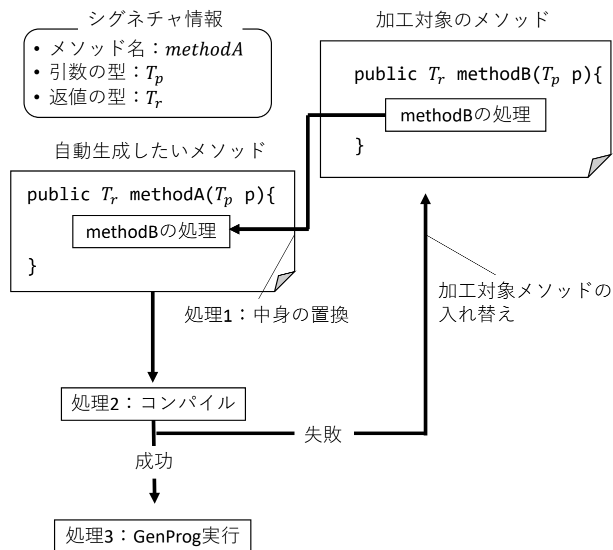

図 4: メソッド加工の処理の流れ

クラス名，プロジェクト名，メソッドの開始行，ソースコード，である．

5.2 メソッドのランク付け

メソッドのランク付けの指標として，メソッド名における類似度を用いる．これは，メソッド名が類似していると，メソッドの処理内容も類似している可能性が高いという先行研究の調査結果に基づいている [13] [14]．入力されたメソッド名と，抽出されたメソッド群のメソッド名との類似度をレーベンシュタイン距離を用いて算出する．その値が小さいほど，優先度が高いと判断する．

5.3 メソッドの加工

メソッドの加工には，自動プログラム修正ツールである GenProg [5] を用いた．図 4 に具体的なメソッドの加工の流れを示す．まず，自動生成したいメソッドと同じシグネチャのメソッド (加工対象メソッドと呼ぶ) を取得する．その後の処理として，以下の 3 つに分かれる．

処理 1 自動生成したいメソッドの中身 (シグネチャ以外) を加工対象メソッドの中身と置き換える．

処理 2 コンパイルが成功すれば処理 3 に移行し，コンパイルが失敗すれば別の加工対象メソッドを用いて処理 1 に戻る．

処理 3 GenProg を実行し，テストケースをすべて通過するメソッドが生成されれば，それを出力する．生成されなければ，別の加工対象メソッドを用いて処理 1 に戻る．

6. 適 用 実 験

本章では，4 つのオープンソースプロジェクトに対して行った適用実験の内容と，その結果について述べる．本実験の目的は，2. で述べたようなメソッドの生成が，プロジェクト全体でどの程度成功するのか調査することである．

6.1 実 験 手 順

実験対象のプロジェクトから，自動生成したいメソッドを 1 つ選択する．その後，5.3 で述べた処理を行い，自動生成を試みる．これを，プロジェクト内に存在する全ての実験対象メソッドに対して行う．

実験には，以下の 4 つのオープンソースプロジェクトを用いた．

- apache-commons-text
- apache-commons-lang
- apache-commons-io
- apache-commons-collections

上記のプロジェクトを選んだ理由は，Java で開発されており，Junit を用いたテストケースも存在するからである．また，プロジェクト内に存在するメソッドのうち，実験対象となるのは以下の条件を満たすメソッドである．

- 加工対象メソッドが 1 つ以上存在
- 命令カバレッジとブランチカバレッジがともに 100%
- ステートメント数が 2 以上
- Junit を用いたテストケースが存在

カバレッジを用いる理由は 6.2 で述べる．ステートメント数が 2 以上のメソッドを用いる理由は，ステートメント数が 1 のメソッドはセッターやゲッターであり，メソッドの自動生成という本手法のコンテキストとは合わないからである．テストケースが存在しないと，提案手法が適用できないためテストケースが必要となる．また，提案手法における Step 2 において，GenProg のタイムアウト時間を 30 分とした．

6.2 テストケースの調査

プロジェクト内に存在するメソッドに対して提案手法を適用するためには，テストケースが必要である．一般的に，テストメソッドには，テスト対象のメソッド名の接頭に'test' という文字列を付加するが，そうでない場合もある．したがって，テストメソッドの名前だけではテストケースが存在するかどうかの判別が困難である．そこで，各メソッドに対応するテストケースが存在するか調査するために，カバレッジを算出した．カバレッジの算出には，Eclipse のプラグインである EclEmma を用いた [15]．このツールでは，メソッドごとの命令カバレッジおよびブランチカバレッジを算出することができる．算出したカバレッジの値が高いものを，実験対象とする．

6.3 実 験 結 果

実験結果を表 1 に示す．加工可能なメソッド数とは，実験対象メソッドのうち，5.3 で述べた処理 2 において，コンパイル可能な加工対象メソッドが 1 つ以上存在するメソッドの数である．自動生成されたメソッド数とは，全てのテストケースを通過するソースコードを生成できたメソッドの数である．正しく自動生成されたメソッド数とは，自動生成されたメソッドからオーバーフィッティングのメソッドを取り除いたものの数である．オーバーフィッティングとは，入力として与えられたテストケースは満たすが，それ以外は満たさない状態のことである．ただし，自動生成されたメソッドがオーバーフィッティングかどうかの判断は，目視確認によるものである．

図 5 および図 6 に，自動生成されたメソッドのコード例を示す．図 5 はオーバーフィッティングの例，図 6 は正しく自動生成された例である．行番号の横の'-' は，削除された加工対象メソッドのコードを表す．'+' は，既存のコードが挿入されたことを表す．図 5 の removeStart メソッドは，文字列 remove

オリジナルコード

```
1 public static String removeStart(final String str,
2 final String remove) {
3     if (isEmpty(str) || isEmpty(remove)) {
4         return str;
5     }
6     if (str.startsWith(remove)){
7         return str.substring(remove.length());
8     }
9     return str;
10 }
```

生成コード

```
1 public static String removeStart(final String str,
2 final String remove) {
3      if ((isEmpty(str)) || (isEmpty(remove))) {
4          return str;
5   }
6-      if (str.endsWith(remove) {
7-         return strstr.substring(0,
           ((str.length()) - (remove.length())));
6+      if (startsWithIgnoreCase(str, remove)) {
7+         return str.substring(remove.length());
8      }
9      return str;
10 }
```

図 5: オーバーフィッティングの例

オリジナルコード

```
1 public static boolean isBlank(final CharSequence cs) {
2        int strLen;
3        if (cs == null || (strLen = cs.length()) == 0) {
4            return true;
5        }
6        for (int i = 0; i < strLen; i++) {
7            if (!isWhitespace(cs.charAt(i))) {
8                return false;
9            }
10        }
11        return true;
12    }
```

生成コード

```
1 public static boolean isBlank(final CharSequence cs) {
2     if (isEmpty(cs))
3-         return false;
3+         return true;
4     final int sz = cs.length();
5     for (int i = 0; i < sz; i++) {
6         if (!isWhitespace(cs.charAt(i)))
7             return false;
8     }
9     return true;
10 }
```

図 6: 自動生成が成功した例

を文字列 str の接頭から取り除くメソッドである．オリジナルコードと生成コードの差異は 6 行目の if 文の条件式である．生成コードは，6 行目で startsWithIgnoreCase メソッドを呼び出している．このメソッドは，startsWith メソッドの処理に，大文字小文字を区別しないという性質を加えたものである．removeStart メソッドに対する既存テストケースには，大文字小文字の区別を考慮していないので，オーバーフィッティングとなっている．次に図 6 では，isBlank メソッドの自動生成を示している．isBlank メソッドは，与えられた文字列が空白かどうか判定するメソッドである．オリジナルコードと生成コードで大きく異なる点は，2 行目の if 文の条件式である．生成コードにおける isEmpty メソッドはオリジナルコードの 3 行目の条件式と等価なメソッドである．したがって，オリジナルコードと生成コードは等価な処理内容となっており，自動生成が成功していることがわかる．

7. 考　　察

自動生成された各メソッドが，ランク付けされた加工対象のメソッドのうち何番目を使用することで生成されたかを調べた．図 7 にその結果を箱ひげ図として示す．箱ひげ図における横軸は，対象プロジェクトのうち，自動生成されたメソッドがな

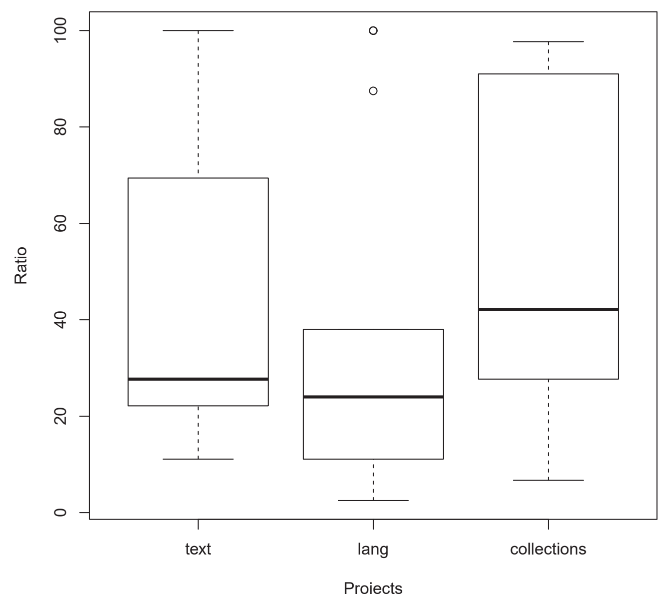

図 7: 自動生成に用いた加工対象メソッドの割合

かった commons-io 以外の 3 プロジェクトを示している．また縦軸は，加工対象のメソッド数に対する加工に用いたメソッド数の割合である．加工に用いたメソッド数はすなわち，5.3 における処理 2 を行った回数である．図から，3 つのプロジェクトにおいて中央値が 50%を下回っていることがわかる．このことから，メソッドのランク付けの手法として，メソッド名の類

表 1: 実験結果

プロジェクト名	実験対象メソッド数	加工可能なメソッド数	自動生成されたメソッド数	正しく自動生成されたメソッド数
commons-text	98	66	7	5
commons-lang	512	415	18	10
commons-io	142	65	0	0
commons-collections	492	168	6	3

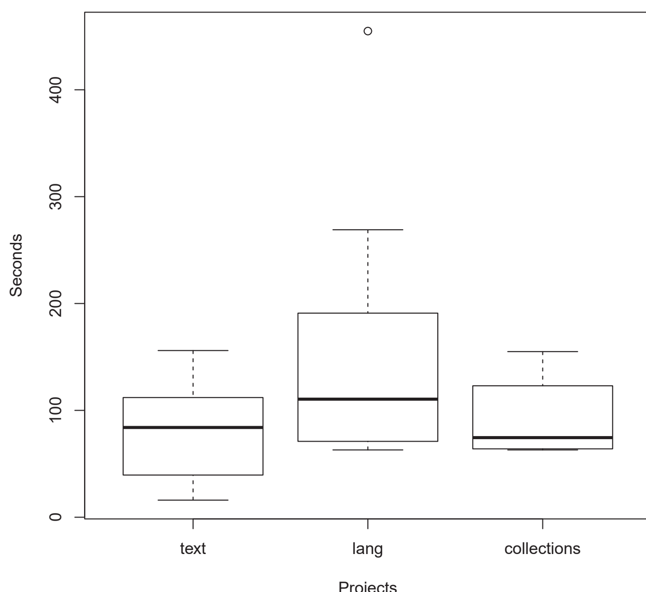

図 8: 自動生成に要した時間

似度は有用であると考えられる．

次に，自動生成に要した時間について考察する．自動生成されなかったメソッドは，30 分のタイムアウトによるものか，全ての加工のパターンを試したが最終的にテストケースを通過しなかったことが要因である．それらを除き，自動生成されたメソッドに要した時間の結果を図 8 に示す．全てのメソッドが，タイムアウト時間よりも大幅に短い時間で生成されていることがわかる．このことから，自動生成の精度を改善するためには，タイムアウト時間を延ばすのではなく，一世代あたりの個体数など，遺伝的アルゴリズムの設定を変更する必要があると考えらえる．

8. おわりに

本研究では，Java メソッドの仕様からソースコードを自動生成する手法を提案した．提案手法では，メソッドのシグネチャ情報と入出力情報を与えることで，目的のメソッドを得ることができる．

提案手法をツールとして実装し，オープンソースソフトウェアに対して適用した．その結果，処理内容が類似したメソッドを加工することによって，自動生成が可能であることが分かった．

今後の取り組みとして，提案手法の評価実験をさらに行っていきたいと考えている．実験の流れとしては，まずプロジェクトの開発履歴を解析する．そして，あるリビジョン r とリビジョン r+1 間において，追加されたメソッドをテストデータとする．つまり，リビジョン r+1 に追加されたメソッドが，リビジョン r 内のメソッドを加工することで自動生成できるかどうかを評価する．

また，GenProg 以外の自動プログラム修正ツールを用いた実装も行う．Xuan らは，プログラム意味論に基づく手法を用いた Nopol を開発した [16]．プログラム意味論に基づく手法は，既存のソースコード中に存在しない記述を用いた加工が可能であることが特徴である．この手法は，欠陥の箇所が満たすべき論理式を導き，その論理式を SMT ソルバを用いて解く．今後の取り組みとして，Nopol を用いて実験を行い，GenProg を用いた場合との比較を行っていきたいと考えている．

謝辞 本研究は，日本学術振興会科学研究費補助金基盤 研究 (S)(課題番号：JP25220003) の助成を得て行われた．

文　献

[1] Armando Solar-Lezama. Program sketching. *International Journal on Software Tools for Technology Transfer*, Vol. 15, No. 5-6, pp. 475–495, 2013.

[2] Tihomir Gvero and Viktor Kuncak. Synthesizing java expressions from free-form queries. *Acm Sigplan Notices*, Vol. 50, No. 10, pp. 416–432, 2015.

[3] Sumit Gulwani, William R Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. *Communications of the ACM*, Vol. 55, No. 8, pp. 97–105, 2012.

[4] 森畑明昌. プログラミングするプログラム-自動プログラム作成最前線. 情報処理, Vol. 57, No. 6, pp. 544–549, 2016.

[5] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, Vol. 38, No. 1, pp. 54–72, 2012.

[6] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan JC Van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, Vol. 82, No. 11, pp. 1780–1792, 2009.

[7] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Software Engineering (ICSE), 2012 34th International Conference on*, pp. 3–13. IEEE, 2012.

[8] Earl T Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. The plastic surgery hypothesis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 306–317. ACM, 2014.

[9] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pp. 24–36. ACM, 2015.

[10] Vidroha Debroy and W Eric Wong. Using mutation to automatically suggest fixes for faulty programs. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pp. 65–74. IEEE, 2010.

[11] Matias Martinez and Martin Monperrus. Astor: a program repair library for java. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pp. 441–444. ACM, 2016.

[12] SQLite. https://www.sqlite.org/.

[13] Anna Corazza, Sergio Di Martino, Valerio Maggio, and Giuseppe Scanniello. Investigating the use of lexical information for software system clustering. In *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*, pp. 35–44. IEEE, 2011.

[14] Yoshiki Higo and Shinji Kusumoto. How should we measure functional sameness from program source code? an exploratory study on java methods. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 294–305. ACM, 2014.

[15] EclEmma. https://www.eclemma.org/.

[16] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime-ClÃĪment, Sebastian Lamelas, Thomas Durieux, Daniel Le Berre, Martin Monperrus. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering*, 2016.