主要処理に着目したメソッド単位のコードクローン検出

長瀬　義大† 石原　知也† 楊　嘉晨† 堀田　圭佑†
肥後　芳樹† 井垣　宏† 楠本　真二†

Method-Based Clone Detection Focused on Core Processes

Yoshihiro NAGASE†, Tomoya ISHIHARA†, Jiachen YANG†, Keisuke HOTTA†, Yoshiki HIGO†, Hiroshi IGAKI†, and Shinji KUSUMOTO†

あらまし　ソフトウェア開発において，多くのソフトウェアで頻繁に利用される機能をまとめたライブラリの利用は，ソフトウェアの信頼性や開発効率の向上に有用である．そのようなライブラリに含めるべき機能を特定するためには，大規模なプロジェクト群に存在するコードクローンを検出することが有益であると考えられる．そのため，このようなコードクローンを高速に検出することを目的としたメソッド単位の検出手法が提案されている．しかし既存手法では，プログラムの主要処理が同じメソッドであっても副次処理が異なるためにコードクローンとして検出されない場合がある．そこで，副次処理に関するコードを除去した上でメソッド単位のコードクローン検出を行う手法を提案する．この手法により，メソッドの主要処理が同じであれば，たとえその中に存在している副次処理が異なっていても一つのライブラリ化候補として検出することが可能になる．

キーワード　コードクローン，ライブラリ作成，大規模データセット，ソフトウェア保守

1. まえがき

ソフトウェア開発において，ライブラリの利用はソフトウェアの信頼性や開発効率の向上に有用である．これは，開発したいプログラムの機能が既にライブラリに含まれている場合，開発者がその機能を実装する必要がないためである．

開発者がこのようなライブラリの恩恵を受けるためには，必要とする機能がライブラリに含まれている必要がある．そのためには，頻繁に利用される機能をライブラリ化しておく必要がある．ライブラリ化すべき機能を特定するためにコードクローン検出手法は有益であると考えられる．

コードクローンとは，ソースコード中に存在する同一，あるいは類似したコード片のことである．ソースコードの再利用などの理由により，複数のソフトウェアにまたがるコードクローンが多数存在することが示されている [1]．現在，このような複数のソフトウェアにまたがるコードクローンを高速に検出することを目的とした手法として，ファイル単位のコードクローン検出手法 [2], [3]，並びにスケーラブルな細粒度コードクローン検出手法 [1], [4]～[8] が提案されている．

ファイル単位のコードクローン（以降ファイルクローン）検出手法 [2], [3] では，ソースコードをファイル単位で比較することによりファイルクローンの検出を行う．しかしこれらの手法では，ファイル単位でしか比較を行わないため，ファイルの一部のみがコードクローンである場合には検出することができない．

スケーラブルな細粒度コードクローン検出手法 [1], [4]～[8] では，大規模なデータセットから細粒度でコードクローン検出を行う．これらの手法はファイルクローン検出手法では検出できない，ファイルの一部のみのクローンを検出することが可能であるが，速度面でファイルベースの手法に劣っている．また，細粒度な検出手法では検出単位が細かすぎるため，機能の一部のみを含むコードクローンを多く検出してしまう．そのため，ファイルベースの手法，細粒度な手法ともにライブラリの作成に利用するには不十分である．

これに対し石原らは，複数のソフトウェアからライブラリ化の候補を高速に検出することを目的とした，メソッド単位の検出手法を提案している [9]．この手法

† 大阪大学大学院情報科学研究科，吹田市
Graduate School of Information Science and Technology, Osaka University, Suita-shi, 565–0871 Japan

では，メソッドを単位として検出を行うため，コードクローン情報から簡単にライブラリ化を行うことができる．また，メソッドからハッシュ値を算出し，そのハッシュ値を用いて検出を行うため，巨大なデータセットに対しても十分高速に実行することが可能である．

しかしこのメソッド単位の手法では，プログラムの目的とする機能が同じであってもエラー処理などの副次的な処理が異なるために，検出されないコードクローンが存在する．プログラムには，目的とする機能を実現するための実装（主要処理）と排除しても目的とする機能実現に影響を及ぼさない主要処理以外の実装（副次処理）の 2 種類が含まれている．そのため，メソッド単位のコードクローン検出手法において副次処理の影響を排除することができれば，ライブラリ化の候補となるメソッドを主要処理のみに着目して検出することが可能となる．そこで本研究では，メソッド単位のコードクローン検出に副次処理の除去を追加しコードクローン検出を行うことで，ライブラリ化候補の特定を目指す．この手法により，メソッドの主要処理が同じであれば，たとえその中に存在している副次処理が異なっていても一つのライブラリ化候補として検出することが可能になる．

2. 主要処理と副次処理

2.1 定　　義

主要処理とはプログラムの目的とする機能を実装した処理であり，**副次処理**とは排除しても目的とする機能実現に影響を及ぼさない主要処理以外の処理である．副次処理はプログラムの耐故障性を向上させるために利用されることが多い [10]．その性質から，副次処理が主要処理に直接影響を与えることは少ないと考えられる．プログラムの全ての処理は主要処理か副次処理のいずれかに属する．

副次処理としてよく利用される処理の一例として，例外処理やロギング処理が挙げられる．例外処理とはプログラムの実行過程で発生する意図しない状況に陥ったときの処理である．例えば，意図しない引数が渡された場合の例外処理は副次処理と考えられる．ロギング処理はプログラムの実行過程を時系列で記録に残すための処理である．主に障害発生時にプログラムの入出力や処理の経緯を確認するために用いられるが，通常主要処理に影響をあたえることはない．

2.2 ライブラリ化における主要処理と副次処理

ライブラリとは，他のプログラムから利用できるように特定の機能をもったプログラムをまとめたものである．開発者はライブラリ化されたプログラムを使用することで必要な機能を新しく実装する必要がなくなり，開発コストを小さくすることができる．また，ライブラリ化されたプログラムは既存のソフトウェアにおいて繰り返し検証されているため，開発者は信頼性の高いプログラムを使用できる．

ライブラリ化されたプログラムを使用する際に，開発者はそのプログラムのもつ機能に注目する．開発者の必要とする機能をもつプログラムを使用しなければ，ライブラリの恩恵を享受できないからである．ここで，先の定義により，プログラムの機能を実装しているのは主要処理である．このことから，ライブラリ化されたプログラムで重要なのは主要処理であり，副次処理はライブラリの便益とは直接関係がないことが分かる．よって，ライブラリ化の判断をする際にはプログラムの主要処理にのみ注目し，副次処理は考慮する必要がないと考えられる．

3. 主要処理のみを考慮したメソッド単位のコードクローン検出

3.1 既存研究におけるメソッド単位のコードクローン検出

石原らは，開発者による共通処理のライブラリ化の支援を目的として，複数のソフトウェアを対象としたメソッド単位のコードクローン検出手法を提案した [9]．石原らの手法の概要を図 1 に示す．また，各 STEP の概要を以下に示す．

STEP1（メソッドの切り出し） 与えられたソースファイルからメソッドを切り出す．メソッドの切り出しは，ソースファイルから作成した抽象構文木上で実現する．抽象構文木を作成することで，改行や空白の違いなどが吸収される．

STEP2（正規化） STEP1 で切り出されたメソッド

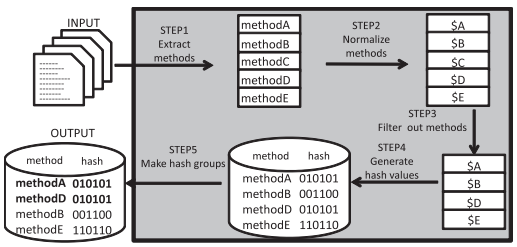


図 1　石原らの手法の概要
Fig. 1　An overview of the existing method.

```
public static void copyFile(File in, File out) {
  if ((in == null) || (out == null)) {
    throw new IllegalArgumentException(
                    "in and out must not be null!");
  }
  try {
    FileInputStream fis = new FileInputStream(in);
    FileOutputStream fos = new FileOutputStream(out);
    byte[] buf = new byte[1024];
    int i = 0;
    while ((i = fis.read(buf)) != -1) {
      fos.write(buf, 0, i);
    }
    fis.close();
    fos.close();
  }catch(Exception e) {
     logger.error("Caught " + e.getClass().getName(), e);
  }
}
```

(a) exception throwing,try-catch,logging

```
static public void CopyFile(File source, File dest)
                                          throws Exception {
  if (source == null || dest == null) {
    throw new IllegalArgumentException();
  }
  pLogger.log(Level.FINEST, "CopyFile",
                "Copying file from " +source.toString()
                  + " to " + dest.toString());
  FileInputStream fis = new FileInputStream(source);
  FileOutputStream fos = new FileOutputStream(dest);
  byte[] buf = new byte[1024];
  int i = 0;
  while ((i = fis.read(buf)) != -1) {
    fos.write(buf, 0, i);
  }
  fis.close();
  fos.close();
}
```

(b) throws clause,exception throwing,logging

```
public void copySingleFile(File in, File out)
                          throws FileCopyException {
  try{
    FileInputStream fis  = new FileInputStream(in);
    FileOutputStream fos = new FileOutputStream(out);
    byte[] buf = new byte[1024];
    int i = 0;
    while((i=fis.read(buf))!=-1) {
      fos.write(buf, 0, i);
    }
    fis.close();
    fos.close();
  } catch (Exception ex) {
    throw new FileCopyException(
                    "IO stream Exception",ex);
  }
}
```

(c) throws clause,exception throwing,try-catch

```
static void copyFile(File in, File out)
                      throws Exception {
  try {
    FileInputStream fis = new FileInputStream(in);
    FileOutputStream fos = new FileOutputStream(out);
    byte[] buf = new byte[1024];
    int i = 0;
    while ((i = fis.read(buf)) != -1) {
      fos.write(buf, 0, i);
    }
    fis.close();
    fos.close();
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

(d) throws clause,try-catch

```
public static void copyFile(File in, File out)
                                 throws Exception{
  System.err.println("copyFile: " + in + " -> " + out);
  FileInputStream fis  = new FileInputStream(in);
  FileOutputStream fos = new FileOutputStream(out);
  byte[] buf = new byte[1024];
  int i = 0;
  while((i=fis.read(buf))!=-1) {
    fos.write(buf, 0, i);
  }
  fis.close();
  fos.close();
}
```

(e) throws clause,System.err.println

```
private void copyFile(File in, File out)
                                 throws IOException {
  FileInputStream fis  = new FileInputStream(in);
  FileOutputStream fos = new FileOutputStream(out);
  byte[] buf = new byte[1024];
  int i = 0;
  while((i=fis.read(buf))!=-1) {
    fos.write(buf, 0, i);
  }
  fis.close();
  fos.close();
}
```

(f) throws clause

図 2　副次処理のみが異なるメソッドの例
Fig. 2　Motivating example.

に対して正規化を行う．行う正規化は，変数名，リテラル，メソッド宣言部のメソッド名の特殊文字列への置換，及び，修飾子，アノテーション，コメント文の削除である．

STEP3（フィルタリング） ソースファイルにはセッターやゲッター等の処理が単純であり，かつ，短いメソッドが多く存在する．このようなメソッドは多くがreturn 文や簡単な代入文のみで構成されているため，大量にコードクローンとして検出されるおそれがある．このようなコードクローンは処理が簡単に記述できるためライブラリ化する価値が小さく，コードクローンとしての検出は不要である．そこで，このようなメソッドはハッシュ値の計算を行わないようにフィルタリングを行う．

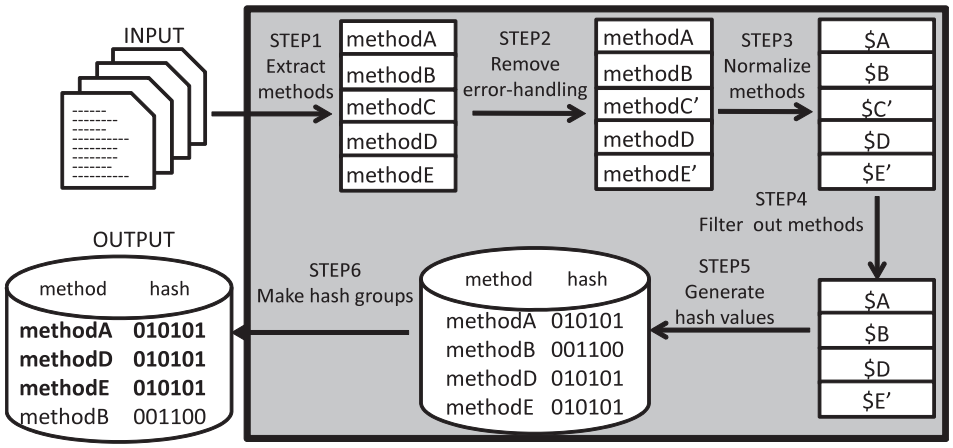


図 3　提案手法の概要
Fig. 3　An overview of the proposed method.

STEP4（ハッシュ値の計算） STEP3 を通過した各メソッドに対してハッシュ値を算出する．具体的には，メソッドを一つの文字列とみなしその文字列に対してハッシュ値の計算を行う．等しい記述をもつメソッドはハッシュ値が等しくなるため，ハッシュ値が等しいメソッド同士はコードクローンの関係にある．ハッシュ値の算出には MD5 [11] を用いており，算出されたハッシュ値はメソッドごとにデータベースに格納する．

STEP5（ハッシュ値によるグループ化） STEP4 で算出されたハッシュ値の等しいメソッドをグループ化する．等しいハッシュ値をもつメソッドは等しい記述をもつ．そのため，ハッシュ値が等しいメソッドをグループ化することで，互いにコードクローンであるメソッド同士が一つのグループを構成する．これにより，コードクローンの関係にあるメソッドがどれか判別できるようになる．

また，石原らは 13,000 以上のソフトウェアからなる UCI source code data sets [2], [12] に対し手法を適用した．その結果，ファイル単位の検出手法では見つけることのできない約 1,160,00 のメソッド単位のコードクローンを検出した．更に，検出されたコードクローンの内，多くのソフトウェアに含まれている 100 のコードクローンを調査した結果，56%のコードクローンがライブラリ化に有用であったと報告している．

3.2　既存研究の問題

石原らの手法を用いることで大規模なデータセットからライブラリ化の候補を特定することができる．しかし，石原らの手法では，主要処理が同じであっても副次処理が異なるために検出されないコードクローンが存在する．副次処理を除去することで検出可能なコードクローンの例を図 2 に示す．これらのメソッドは実際のオープンソースプロジェクトに存在しているメソッドである．これらのメソッドの違いは，各メソッドにおいてハイライトで示している副次処理のみであり，副次処理は異なるが主要処理は等しい．そのため，ライブラリ化を行う候補を特定する際には，これらのメソッドを一つの候補として検出すべきである．しかし石原らの手法では単純な正規化しか行わないため，これらをコードクローンとして検出することができない．そこで，副次処理に関するコードを除去した上でメソッド単位のコードクローン検出を行う．

3.3　既存研究の改善案

本研究では石原らの手法をもとに副次処理の除去を行うメソッド単位のコードクローン検出手法を提案する．そのため提案手法は石原らの手法と等しく，対象とするソースファイル群を入力として受け取り，検出されたコードクローンの情報を出力として返す．本手法の概要を図 3 に示す．

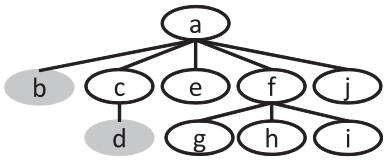
本研究では，正規化処理を行う前に副次処理の除去を行う．副次処理の除去は抽象構文木を利用して行う．石原らの手法ではメソッドの切り出しに抽象構文木を使用しており，メソッドごとに抽象構文木が存在する．このとき，メソッド内の各処理は全て抽象構文木におけるノードとして表現される．そのため，副次処理と判断された処理を表す抽象構文木上のノードを削除することでメソッド内にある副次処理の除去を実現でき

```
a. public static byte[] OR(byte[] a,byte[] b)
b.   throws Exception
     {
c.    if(a.length != b.length) {
d.       throw new Exception();
      }
e.    byte[] v = new byte[a.length];
f.    for(int i=0; i<v.length; i++) {
g.       int A=(a[i]<0 ? a[i]+256 : a[i]);
h.       int B=(b[i]<0 ? b[i]+256 : b[i]);
i.       v[i]=(byte)(B | A);
      }
j.    return(v);
   }
```

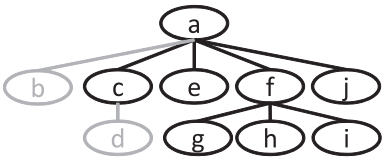
(a) 除去前のコード

```
a. public static byte[] OR(byte[] a,byte[] b)
     {
e.    byte[] v = new byte[a.length];
f.    for(int i=0; i<v.length; i++) {
g.       int A=(a[i]<0 ? a[i]+256 : a[i]);
h.       int B=(b[i]<0 ? b[i]+256 : b[i]);
i.       v[i]=(byte)(B | A);
      }
j.    return(v);
   }
```

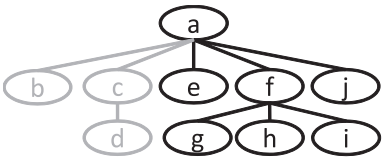
(b) 除去後のコード



(c) 除去前の抽象構文木



(d) throw 文，throws 節を除去



(e) 空になった if 文を除去

図 4　throw 文，throws 節の除去例
Fig. 4　An example of removing a statement throwing an exception and a throws clause.

る．提案手法は石原らの手法で使用されている抽象構文木を使っているため，検出時間に大きな影響を与えずに副次処理の除去を行うことが可能である．

図 4，5 に Java 言語における副次処理除去の例を示す．図 4 では，throw 文と throws 節を副次処理と判断し，除去を行っている．図 4(c)～図 4(e) に抽象構文木上での除去の流れを示す．ここで，抽象構文木の各ノードは図 4(a) の各行と対応している．まず，図 4(c) から throw 文と throws 節に対応するノードが除去される（図 4(d)）．このノードの除去により，if 文を表すノードから子ノードがなくなるために除去を行い（図 4(e)），除去後のコードとなる．

図 5 では，try 文を副次処理と判断し，除去を行っている．図 5(c)～図 5(e) に抽象構文木上での除去の流れを示す．ここで，抽象構文木の各ノードは図 5(a) の各行と対応している．まず，図 5(c) から副次処理として catch 節内部のノードが除去される（図 5(d)）．次に，try 文を表すノードの親ノード（ノード a）を try 節及び finally 節の各子ノードの新しい親ノードとして設定する（図 5(e)）．その後，try 文に対応するノードの除去を行い（図 5(f)），除去後のコードとなる．

4. 副次処理に対する予備調査

4.1 調査の概要

本研究では，副次処理としてよく利用される処理にはどのような処理があるかについて予備調査を行った．調査方法は，一つのオープンソースプロジェクトにある全てのメソッドを著者が目で確認し，プログラムの各処理を主要処理か副次処理のいずれかに分類するというものである．対象となるプロジェクト名は jNetStream，ファイル数は 218，総メソッド数は 749 である．ソースコードは全て Java 言語で記述されている．

4.2 調 査 結 果

調査の結果，以下に示す処理が副次処理に分類された．

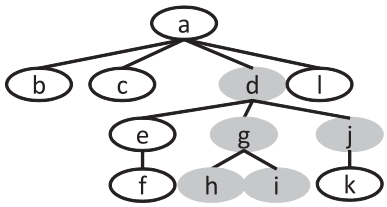
- throw 文
- try～catch～finally と catch 節内部の処理
- throws 節
- return null
- assert 文
- Logger クラスに含まれるメソッド呼び出し
- 標準出力への出力
- 標準エラー出力への出力

```
a. public String read(){
b.    readLock.lock();
c.    String data=null;
d.    try {
e.       while(hasNext()){
f.          data= new String(doRead());
         }
g.    } catch(Exception e){
h.       e.printStackTrace();
i.       data=null;
j.    } finally {
k.       readLock.unlock();
      }
l.    return data;
   }
```

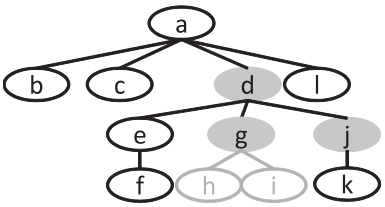
(a) 除去前のコード

```
a. public String read(){
b.    readLock.lock();
c.    String data=null;
e.    while(hasNext()){
f.       data= new String(doRead());
      }
k.    readLock.unlock();
l.    return data;
   }
```

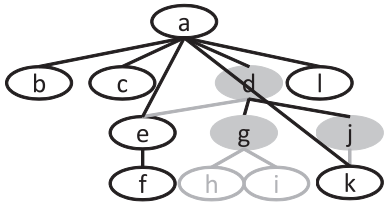
(b) 除去後のコード



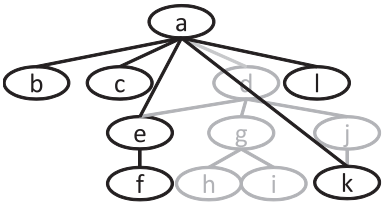
(c) 除去前の抽象構文木



(d) catch 節内部を除去



(e) 親ノードの変更



(f) try～catch～finally 文を除去

図 5　try～catch～finally 文の除去例
Fig. 5　An example of removeing try-catch-finally.

- System.exit() メソッド

本研究では，上記の処理をライブラリ化を考慮する際に除去すべき副次処理と判断した．

5. 副次処理除去の精度の評価

5.1 実験の概要

本実験では，提案手法の適用により，開発者が副次処理として記述した処理をどの程度除去できているかを調査する．本実験ではまず被験者となる開発者に，自身が過去に開発したプログラムに存在するメソッドを提示し，副次処理として記述した箇所を選択してもらう．その後，各メソッドに対し本手法の副次処理除去を適用し，除去された箇所と被験者が副次処理として提示した箇所を比較することで，除去の適合率と再現率を調査する．

本実験の被験者はコンピュータサイエンスを専攻している大学院生 4 名である．各被験者にはそれぞれ 10 のメソッドを提示し，合計 40 のメソッドに対して副次処理除去の精度を調査した．

5.2 実験結果

被験者によって選択された 40 のメソッドに含まれる副次処理と，本手法によって除去された処理がどの程度一致するかを調査した．このとき，一つのメソッドに複数の副次処理が存在した場合は，副次処理ごとに調査を行った．本手法の副次処理除去により，被験者が副次処理として記述した 30 の処理の内，25 の副次処理を除去することができた．適合率は 78.1%，再現率は 83.3%である．

被験者が副次処理として記述したが，除去できなかった例を図 6 に示す．このメソッドにおいて，被験

```
public String unzipAndRegist(File file,
       String projectName, String labelName)
                          throws IOException {
  try {
    uncompress.uncompress(file.getCanonicalPath());
    String savePath = file.getCanonicalPath().substring(0,
               file.getCanonicalPath().lastIndexOf(sep));
    synchronized (session) {
      session.setAttribute("info",
                 "uploaded file:" + savePath);
      session.setAttribute("totalFileNum",
            countFiles(new File(savePath)));
    }
    SQLHelper.getSQLHelper().setSavepoint();
    registToDB(new File(savePath), projectName, labelName);
    …
```

図 6　副次処理の除去に失敗したメソッド
Fig. 6　An example of false negative.

者はハイライトされた箇所を副次処理として記述している．本来であれば，対象プログラムの開発者である被験者がプログラムの目的とする機能に直接影響しないと判断している処理は副次処理に分類されるべきである．しかし本手法では予備調査の結果に基づき除去の対象となる処理を選択していたため，setAttributeを用いたメッセージ送信を副次処理とみなしていない．そのため，この処理を副次処理として除去することができなかった．このように除去できなかった処理または誤って除去してしまった処理が発生した原因は，全ての場合において予備調査の結果と被験者との間で副次処理の判断基準が異なることによるものだった．

6. 石原らの手法との比較

6.1 実験の準備

提案手法の有用性を示すために以下の三つの項目を調査する．

調査項目 A　副次処理の除去を行う場合と行わない場合で，検出されるコードクローンにどの程度違いがあるか

調査項目 B　提案手法と石原らの手法でライブラリ化の候補となるコードクローンにどの程度違いがあるか

調査項目 C　副次処理の除去を行う場合と行わない場合で検出時間がどの程度異なるか

本研究では実験対象として，石原らが実験に用いた”UCI source code data sets” [2], [12] を用いた．また，石原らと同様にデータセット内で trunk，tags，branches を同時に含むソフトウェアは trunk 内のソースファイルのみを，バージョンの異なる同一のソフトウェアは最新バージョンのソフトウェアのみを検出対象とし，ソースコード自動生成ツールによって生成されたファイルを検出対象から除外した．前述の処理を行うことで，石原らが用いたデータセットと等しいデータセットを用いることができる．本実験における実験対象の構成を表 1 に示す．本実験は CPU 数 2，計 8 コア（2.27GHz），メモリ 32GByte の環境で行い，検出結果を格納するデータベースは SSD 上に作成した．また，追加した除去処理の影響のみを調査するため，石原らの手法と提案手法においてフィルタリング処理条件を 30 トークン以下のメソッドを除去するように統一した．

表 1　実験対象の構成
Table 1　An overview of the target data set.

ファイル数	2,072,490
ソフトウェア数	13,193
メソッド数	19,416,603
総行数	361,663,992
全容量	30.6GByte

本実験おいて，クローンセットに含まれるメソッドが分布しているソフトウェアの数を NoS(the Number Of Software system) と定義する．ここで，クローンセットとは互いにコードクローン関係にあるメソッドの集合のことである．多くのソフトウェアで使用されているメソッドは，クローンセットとして検出されたときに NoS の値が大きくなる．したがって，NoS の値が大きいクローンセットはライブラリ化すべきであると考えられる．

また，クローンセットに含まれる各メソッドに対して代入が行われたフィールド数を算出し，その最大値を NAF(the Number of Assignment Fields) として定義する．また，クローンセットに含まれるメソッドに返り値が存在する場合，NAF 値に 1 を加算する．NAF 値が 2 以上のクローンセットは，メソッドの呼び出し元に対して複数の値を返さなければならないため，メソッド単体で抽出することが難しい．それに対して，NAF が 0 若しくは 1 の場合は，メソッド単位で抽出することが可能であり，容易にライブラリ化を行うことができると考えられる．

6.2 調査項目 A：副次処理除去の有無による検出結果の比較

副次処理の除去によりハッシュ値が変化したメソッドがどの程度存在するかを調査した．調査結果を表 2 に示す．ここで，調査したメソッドは副次処理を含んだ状態でトークン数 30 以上のメソッドである．

この結果より，約 45%のメソッドが何らかの副次処理を行っていることが分かる．また，副次処理の除去

表 2　副次処理除去によるメソッドへの影響
Table 2　Impact on methods by removing error-handling.

ハッシュ値が変化したメソッド数	3,403,104(45.8%)
ハッシュ値が変化しなかったメソッド数	4,022,793(54.2%)

表 3　副次処理除去によるクローンセットへの影響
Table 3　Impact on clonesets by removing error-handling.

新たに検出されたクローンセット数	16,820
要素数が増えたクローンセット数	19,938
消失したクローンセット数	69,218

により，プログラムの主要処理は同じであるが副次処理の異なるメソッド群を，一つのクローンセットとして検出できるようになった．これにより，82,639 のメソッドが新たにコードクローンとして検出された．

また，石原らの手法によって検出された全てのクローンセットからメソッドを一つずつ選択し，そのメソッドが提案手法の検出結果に含まれるどのクローンセットに分類されているかを調査した．これによって，石原らの手法では異なるクローンセットに含まれていたが提案手法では共通のクローンセットに含まれるようになったメソッドや，そもそも石原らの手法では検出されていなかったクローンセットがどの程度存在するかを確認できる．調査結果を表 3 に示す．

この結果より，石原らの手法では検出できなかったが，提案手法では検出できたクローンセットが 16,820 存在したことが分かる．このことから，提案手法は石原らの手法では見過ごされていたライブラリ化の候補となるメソッドを検出できるといえる．また，石原らの手法では複数に分かれて検出されるが提案手法では一つのクローンセットに統合されるクローンセットが 19,938 存在した．このことから，副次処理を除去することで，石原らの手法では分割されていた多くのクローンセットが提案手法によって統合されたと考えられる．

ここで，石原らの手法では検出されていたが提案手法では検出されなくなったクローンセットが 69,218 存在する．これは副次処理を除去したためにメソッドのトークン数が減少し，フィルタリングを通過しなくなったためである．

6.3　調査項目 B：ライブラリ化の候補となるコードクローンの違い

提案手法と石原らの手法で検出されたコードクローンにどの程度違いがあるかの調査を行った．しかし，検出されるクローンセットの数が膨大になるため，全てのクローンセットを分析することは現実的に困難である．そこで，提案手法によって検出された NAF 値が 1 以下のクローンセットを NoS 値で降順にソートし，上位のクローンセットから一つずつライブラリ化すべきかどうかを判断した．ライブラリ化の判断を行う際は主要処理にのみ注目した．すなわち，ライブラリ化する上でどの副次処理を含めるべきかについては考慮していない．この作業を繰り返し，最終的に 100 のクローンセットに対してライブラリ化の判断を行った．

ライブラリ化を判断する際，自動生成されたメソッドやコンストラクタ，メインメソッドから構成されるクローンセットは判断の対象から除外した．本研究では実験に使用したデータセットから自動生成されたソースファイルを除いているが完全ではなく，自動生成されたメソッドがコードクローンとして検出されている．しかし，自動生成されたメソッドは同じツールを使うことで再現できるため，ライブラリ化の効果は小さい．コンストラクタは通常インスタンスの初期化を行うといった用途で使用されるため，コンストラクタのみを見てライブラリ化の判断を行ってもライブラリ化の効果を十分に得られない．メインメソッドはソフトウェアごとに独自のものが必要とされるため，ライブラリ化の効果が小さい．このように，いずれの場合もライブラリ化することの利点が小さいため，ライブラリ化の判断を行っていない．

図 7 は，ライブラリ化の判断を行ったクローンセットの NoS 値の変化を表している．ライブラリ化の判断を行ったクローンセットを調査項目 A と同様の方法，すなわち，石原らの手法の検出結果から一つメソッドを選択し，そのメソッドが提案手法の検出結果のどのクローンセットに分類されているかを調べて比較した．図 7 の縦軸は NoS 値，横軸はライブラリ化の判断を行ったクローンセットを示す．グラフ上の色の濃い部分は，提案手法によって増加した NoS 値を示している．図 7 より，調査したクローンセットのうち 39%のクローンセットは石原らの手法から NoS 値が増加していることが分かる．中には，NoS 値が 2 倍以上になったクローンセットも存在した．このことから提案手法は，既存手法では発見できなかった多くのソフトウェアにまたがって存在する処理を発見していることが分かる．

本実験では，石原らと同様に，ライブラリ化の判断

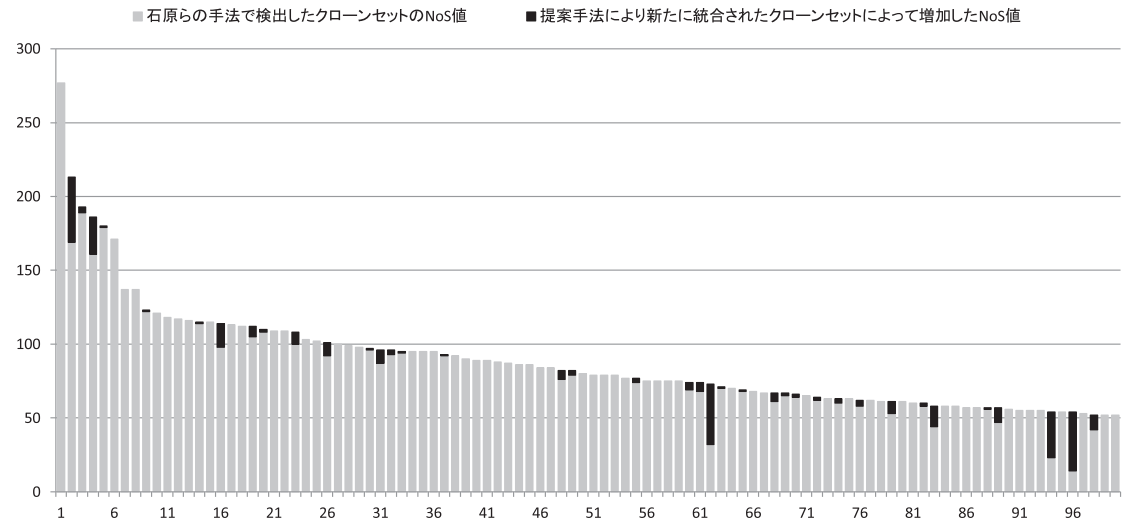


図 7　調査した 100 クローンセットの NoS 値の変化
Fig. 7　Change in the NoS value of 100 clone sets.

```
public void  sort(Object sender)
{
   checkModel();
   compares = 0;
   shuttlesort((int[])indexes.clone(),
                indexes, 0, indexes.length);
   System.out.println("Compares: "
                   +compares);
}
```

(a) sun

```
private void sort(Object sender)
{
   checkModel();
   compares = 0;
   shuttlesort((int[])indexes.clone(),
                indexes, 0, indexes.length);
   logger.debug("Compares: "
                   + compares);
}
```

(b) BasicQuery

図 8　ライブラリ化すべきと判断したコードクローンの例
Fig. 8　An instance of method cloneset suite for library building.

```
public static String
       normaliseNewlines(String input)
{
   input = input.replaceAll("¥r¥n", "¥n");
   input = input.replaceAll("¥r", "¥n");
   return input;
}
```

(a) jbootcat

```
public static String
           convertToEscapes(String message)
{
   message = message.replaceAll("<", "&lt;");
   message = message.replaceAll(">", "&gt;");
   return message;
}
```

(b) YAWL

図 9　ライブラリ化すべきでないと判断したコードクローンの例
Fig. 9　An instance of method cloneset don't suite for library building.

を行った 100 のコードクローンの発生原因を以下の三つに分類した．

- 抽象クラスの継承，インターフェイスの実装
- ソースコードの流用
- 汎用的な処理を行うメソッド

その結果を表 4 に示す．ここで合計が 100 を超えているが，これは発生原因に重複があるためである．本実験では石原らと同様に汎用的な処理を行うメソッドに分類されたクローンセットを除く，65 のクローンセットをライブラリ化すべきであると判断した．石原らの手法で検出したクローンセットに対して同様の調査を行ったところ，調査した 100 クローンセットのうち 62

表 4　クローンセットの分類結果
Table 4　Classification of the clonesets.

抽象クラスの継承，インターフェイスの実装	17
要素数が増えたクローンセット数	64
汎用的な処理を行うメソッド	35

のクローンセットがライブラリ化すべきであると判断された．石原らの手法の調査対象と提案手法の調査対象では，NoS 値の増加によって 10 のクローンセットが入れ替わっており，そのうち三つが新たにライブラリ化すべきであると判断された．この結果から，提案手法は石原らの手法では下位に存在したライブラリ化すべきコードクローンを上位に位置づけることができるといえる．

図 8 にライブラリ化すべきであると判断したメソッドの一例を示す．この二つのメソッドは，ともに JTable のソートを実行するメソッドであり，ハイライトされた箇所のロギング処理のみが異なっている．このメソッドを含むクローンセットは，96 のソフトウェアにまたがって存在する．このような処理は，今後も多くのソフトウェアで行われる可能性があるため，ライブラリ化を行うべきであると判断した．

また，ライブラリ化には適していないと判断したコードクローンの一例を図 9 に示す．この二つのメソッドはユーザ定義名と文字列リテラルのみが異なるためコードクローンとして検出されている．このようなメソッドを含むコードクローンは多くのソフトウェアに存在するが，処理が単純であり，かつ，ソフトウェアごとに異なる目的で用いられる可能性が高いためライブラリ化には適さない．

6.4　調査項目 C：実行時間

作成したツールを対象のデータセットに適用した結果，副次処理の除去を行う場合は 1 時間 50 分，行わない場合は 1 時間 51 分をそれぞれ要した．表 5 に各処理に要した時間を示す．

副次処理の除去を行う場合と行わない場合での実行時間を比較すると，除去を行う場合ではメソッドの切り出しからハッシュ値の計算までの処理にわずかに時間がかかっている．これは，副次処理の除去を追加しているためであるが，時間の増加はわずかであり，この処理にはほとんど時間がかかっていない．また，ハッシュ値のグループ化に要する時間が減少している．これは，提案手法では副次処理の除去後にフィルタリングを行っているため，副次処理除去後にフィルタリング条件を通過できないメソッドが発生し，対象となるメソッド数が減少したためである．実験結果より，作成したツールでは，約 360,000,000 行のソースコードに対し 2 時間以内で検出を行うことができ，実用的な時間で実行可能であるといえる．

表 5　解 析 時 間
Table 5　Analyzing time.

処理内容	除去あり	除去なし
メソッドの切り出し 副次処理の除去 正規化 フィルタリング ハッシュ値の計算	1 時間 29 分	1 時間 28 分
ハッシュ値によるグループ化	21 分	23 分
合計	1 時間 50 分	1 時間 51 分

7.　妥当性への脅威

7.1　副次処理の判断基準に関して

本研究では，除去すべき副次処理を **4.** で記述した予備調査をもとに判断している．しかし，**5.** の実験結果から分かるように，どの処理を副次処理とみなすかは判断する人によって異なる．そのため，本研究で対象とした副次処理を変更することで結果が変わる可能性がある．

また，本研究では副次処理の除去後にフィルタリング処理を行っているため，副次処理の除去後にフィルタリング条件を満たすようになったメソッドは検出対象から除外される．そのため，副次処理の基準が変われば，副次処理の除去後にフィルタリング条件を満たすメソッドも変化する．これにより，検出対象となるメソッドが変化し，結果が変わる可能性がある．

7.2　ハッシュ値の衝突に関して

提案手法では二つのメソッドがコードクローンであるかの判定にハッシュ値を使用している．そのためコードクローン検出の際に文字列が異なるにもかかわらずハッシュ値が偶然一致するハッシュ値の衝突が起こった場合，本来コードクローンでないメソッドがコードクローンとして検出されるおそれがある．しかし，本研究ではハッシュ値の計算に 128bit の MD5 アルゴリズムを使用しているため衝突確率は極めて低いと考えられる．また本研究では，同じハッシュ値をもつメソッド全てについて，副次処理の除去と正規化処理を行った後の文字列が一致するかどうかを調べるプログラムを作成し，ハッシュ値の衝突がなかったことを確認した．ただし，より多くのソフトウェアに対し

提案手法を適用した場合，ハッシュ値の衝突が起こる可能性がある．

7.3 正規化，フィルタリングに関して

本研究は石原らの手法をもとにしており，以下の正規化処理を行っている．

- 変数名，リテラル，メソッド宣言部のメソッド名は特殊文字列に置換する．
- 修飾子，アノテーション，コメント文は削除する．

しかし，型名の正規化の有無やフィルタリングを行うトークン数の変更等により，本研究では検出できていないコードクローンを検出できる可能性がある．

8. むすび

本論文では，副次処理のみが異なるメソッド群を一つのライブラリ化候補として検出することを目的として，副次処理の除去を追加したメソッド単位のコードクローン検出手法を提案し，評価を行った．今後の課題として，

- 主要処理や副次処理の分類に関して更に詳しい調査を行い，除去対象となる処理を変えることでどの程度違いがあるか調査
- 本研究とは異なる正規化やフィルタリング処理を行った場合に，検出されるコードクローンにどの程度違いがあるかの調査
- 対象となるファイルを Java 以外にも拡張し，言語によって検出されるコードクローンに違いがあるかの調査

等が挙げられる．

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：25220003），挑戦的萌芽研究（課題番号：24650011），及び文部科学省科学研究費補助金若手研究 (A)（課題番号：24680002），独立行政法人情報処理推進機構　技術本部　ソフトウェア高信頼化センター（SEC: Software Reliability Enhancement Center）が実施した「2012 年度ソフトウェア工学分野の先導的研究支援事業」の支援を受けた．

文　献

[1] 肥後芳樹，リビエリシモネ，松下　誠，井上克郎，“大規模ソースコードを対象としたコードクローンの検出と可視化，” 情処学論，vol.48, no.11, pp.3510–3519, Nov. 2007.

[2] J. Ossher, H. Sajnani, and C. Lopes, “File cloning in open source java projects: The good, the bad, and the ugly,” Proc. 27th International Conference on Software Maintenance, pp.283–292, Sept. 2011.

[3] 佐々木裕介，山本哲男，早瀬康裕，井上克郎，“大規模ソフトウェアシステムを対象としたファイルクローンの検出，” 信学論（D），vol.J94-D, no.8, pp.1423–1433, Aug. 2011.

[4] J.R. Cordy and C.K. Roy, “Debcheck: Efficient checking for open source code clones in software systems,” Proc. 19th International Conference on Program Comprehension, pp.217–218, June 2011.

[5] Y. Higo, K. Tanaka, and S. Kusumoto, “Toward identifying inter-project clone sets for building useful libraries,” Proc. 4th International Workshop on Software Clones, pp.87–88, May 2010.

[6] B. Hummel, E. Jürgens, L. Heinemann, and M. Conradt, “Index-based code clone detection: incremental, distributed, scalable,” Proc. 26th International Conference on Software Maintenance, pp.1–9, 2010.

[7] R. Koschke, “Large-scale inter-system clone detection using suffix trees,” Proc. 16th European Conference on Software Maintenance and Reengineering, pp.309–318, March 2012.

[8] W. Shang, B. Adams, and A.E. Hassan, “An experience report on scaling tools for mining software repositories using mapreduce,” Proc. 25th International Conference on Automated Software Engineering, pp.275–284, Sept. 2010.

[9] 石原知也，堀田圭佑，肥後芳樹，井垣　宏，楠本真二，“大規模なソフトウェア群を対象とするメソッド単位でのコードクローン検出，” 情処学論，vol.54, no.2, pp.835–844, Feb. 2013.

[10] C. Fu and B.G. Ryder, “Navigating error recovery code in java applications,” Proc. OOPSLA workshop on Eclipse Technology eXchange, pp.40–44, Oct. 2005.

[11] R. Rivest, “The md5 message-digest algorithm,” RFC 1321 (Informational), April 1992. Updated by RFC 6151. http://www.ietf.org/rfc/rfc1321.txt

[12] C. Lopes, S. Bajracharya, J. Ossher, and P. Baldi, “Uci source code data sets”. http://www.ics.uci.edu/lopes/datasets/

（平成 25 年 2 月 6 日受付，6 月 4 日再受付）

長瀬　義大

平成 23 年大阪大学基礎工学部情報科学科卒業．平成 25 年同大学大学院情報科学研究科博士前期課程修了．在学時，コードクローン分析に関する研究に従事．

石原　知也

平成 24 年大阪大学基礎工学部情報科学科卒業．現在，同大学大学院情報科学研究科博士前期課程在学中．コードクローン分析に関する研究に従事．

楊　　嘉晨

平成 23 年中国上海交通大学軟件学院軟件工程専業卒業．平成 25 年大阪大学大学院情報科学研究科博士前期課程修了．現在，同研究科博士後期課程在学中．コードクローン分析に関する研究に従事．

堀田　圭佑

平成 22 年大阪大学基礎工学部情報科学科卒業．平成 24 年同大学大学院情報科学研究科博士前期課程修了．現在，同研究科博士後期課程在学中．日本学術振興会特別研究員 DC．コードクローン分析に関する研究に従事．IEEE，情報処理学会各会員．

肥後　芳樹　（正員）

平成 14 年大阪大学基礎工学部情報科学科中退．平成 18 年同大学大学院博士後期課程修了．平成 19 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．博士（情報科学）．ソースコード分析，特にコードクローン分析やリファクタリング支援に関する研究に従事．情報処理学会，日本ソフトウェア科学会，IEEE 各会員．

井垣　　宏　（正員）

平成 12 年神戸大学工学部電気電子工学科卒業．平 14 年奈良先端科学技術大学院大学情報科学研究科博士前期課程修了．平成 17 年同大学院博士後期課程修了．同年同大学院特任助手．平成 18 年南山大学数理情報学部講師．平成 19 年神戸大学大学院工学研究科特命助教．平成 22 年東京工科大学コンピュータサイエンス学部助教．平成 23 年大阪大学大学院情報科学研究科特任准教授．博士（工学）．ソフトウェア工学教育，サービス指向アーキテクチャ，ホームネットワークシステム，Web サービス，ソフトウェアプロセス等の研究に従事．IEEE，ACM，IPSJ 各会員

楠本　真二　（正員：シニア会員）

昭和 63 年大阪大学基礎工学部卒業．平成 3 年同大学大学院博士課程中退．同年同大学基礎工学部助手．平成 8 年同講師．平成 11 年同助教授．平成 14 年同大学大学院情報科学研究科助教授．平成 17 年同教授．博士（工学）．ソフトウェアの生産性や品質の定量的評価に関する研究に従事．情報処理学会，IEEE，IFPUG 各会員．