異常処理を考慮したメソッド単位のコードクローン検出

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

大阪大学 大学院情報科学研究科 コンピュータサイエンス専攻

E-mail: {y-nagase, k-hotta, t-ishihr, higo, igaki, kusumoto}@ist.osaka-u.ac.jp

概要

ソフトウェア間にまたがるコードクローンを検出することは，多くのプロジェクトに頻出する処理のライブラリ化による開発効率の向上やライセンスに違反したソースコード流用の特定などの観点から有益である．そのため，このようなコードクローンを高速に細粒度で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，プログラムの機能が同じメソッドであっても異常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，異常処理に関するコードを除去したうえ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，メソッドの機能が同じであれば，たとえその中に存在している異常処理が異なっていても *1* 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．

1. はじめに

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すで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，複数のプロジェクトにおいて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．このようなライブラリの候補を検出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．ソースコードの流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が多く提案されている．

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を検出する．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][5][6][7][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と比較して多くのコード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．

ファイルクローン検出手法，スケーラブルな細粒度コードクローン検出手法はともに複数のプロジェクトからのコードクローン検出を行うことが可能である．しかし，ファイルクローン検出手法では検出単位が大きすぎるため，コードクローンの見逃しが発生してしまう．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，これらの手法はライブラリの作成に利用するには不十分である．

そのため，複数のプロジェクトからライブラリ化に有用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 [9]．この手法では，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，メソッドをハッシュに変換して検出を行うため，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，プログラムの機能が同じであっても異常処理の違いにより検出されないコードクローンが存在する．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異常処理の除去を追加し，コードクローン検出を行う．この手法により，メソッドの機能が同じであれば，たとえその中に存在している異常処理が異なっていても 1 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. 準備

2.1. 異常処理

異常処理とはプログラムの耐故障性の向上や異常な状態の発見に利用される処理である [10][11]．異常処理はプログラムの機能に影響を与えないため，機能が同じメソッドであっても異常処理の実装は複数考えられる．そのため，ライブラリ化を行う候補を検出する際には，異常処理が異なるメソッドであっても機能が同じであれば 1 つの候補として検出すべきである．

本研究は以下の処理を異常処理として扱う．

- 例外のスロー
- 例外処理
- アサーション
- ロギング処理
- プログラムの終了処理

3. 既存研究

3.1 ファイルクローン

佐々木らは，ファイルクローン検出ツール FCFinder を開発し，ファイルクローンの性質を調査した [2]．FCFinder はコメント文の削除や字句解析を行った後，ファイルからハッシュ値を計算しファイルクローンを検出する．また，佐々木らは総行数約 400,000,000 行の大規模なソフトウェア群である FreeBSD Ports Collection に対し FCFinder を適用した．その結果，17 時間ほどで検出を終了し，FreeBSD Ports Collection の約 68%がファイルクローンであったことを報告している．また，検出されたファイルクローンのうち 27%はコメント文やインデントの違いであり，ファイルサイズの分布はファイルクローンであるファイルとそうでないファイルとで差異がなかったとも報告している．

Ossher らは，exact，FQN，fingerprint の 3 つの要素を組み合わせたファイルクローン検出手法を提案し，ファイルクローンが発生する状況を調査した [3]．exact は，各ソースファイルを 1 つの文字列とみなし，その文字列からハッシュ値を計算し，ハッシュ値が一致したファイルをファイルクローンとして検出する．FQN は，クラスの完全限定名が一致しているファイルをファイルクローンとして検出する．fingerprint は，ソースファイル中のメソッド名とフィールド名がどの程度等しいかを調査し，ある閾値を超えて一致しているファイルをファイルクローンとして検出する．Ossher らは，約 13,000 の Java ソフトウェアに対し上記の 3 つの要素を組み合わせて実験したところ，全ファイルの約 10%超がファイルクローンとして検出されたと報告している．またファイルクローンが発生する状況として，同じライブラリを使用していることや新たなソフトウェア開発を始めるときにそれ以前に開発されたソフトウェアの再利用を行うことなどが挙げられるとも報告している．

ファイルクローン検出手法では，ファイルを単位として比較を行うため高速な検出が可能である．しかし，ファイルの一部がコードクローンである場合は検出できないため，頻出する処理のライブラリ化を行う場合には，多くの見落としが発生してしまうと考えられる．

3.2 スケーラブルな細粒度コードクローン検出

Hummel らは，index を用いた増分的コードクローン検出手法を提案した [6]．彼らの手法は，ソースコード中の連続する文に対して index を計算し，その比較を行うことでコードクローンを検出する．また，彼らは 100 台のコンピュータを使用することで，730,000 行のソースコードから 36 分でコードクローンの検出が行えると報

```
1 private int OnChanged(int dispID){
2    if(eventTable==null)
3       return COM.S_OK;
4    OleEvent event = new OleEvent();
5    event.detail = OLE.CHANGED;
6    try {
7        notifyListener(dispID,event);
8    } catch (Throwable e) {}
9    return COM.S_OK;
10}
```

```
1 private int OnChanged(int dispID){
2    if(eventTable==null)
3       return COM.S_OK;
4    OleEvent event = new OleEvent();
5    event.detail = OLE.CHANGED;
6    notifyListener(dispID,event);
7    return COM.S_OK;
8 }
```

図 1. motivating example

告している．

Cordy らは，入力ソースファイルと Debian のソースコード間でコードクローンを検出するためのツール DebCheck を作成した [7]．DebCheck は 1 度だけ 10 時間の準備が必要であるが，準備後は数分で入力ソースファイルと Debian のソースコード間でコードクローンを検出することが可能である．

Koschke はライセンスに違反したコードを検出することを目的として，suffix tree を用いた手法を提案した [8]．また実験により，この手法が index を用いた検出手法よりも高速に検出が可能であることを確認している．

これらの手法は大規模なデータセットに対し，実用的な時間で細粒度なコードクローンを検出することができる．しかし，検出されるコードクローンの数が膨大であり，また，プログラムの機能の一部のみからなるコードクローンが多く検出される．そのため，これらの手法をライブラリの作成に利用することは難しい．

3.3 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]．この手法では，まず，対象となるファイルからメソッドを取り出し，コメント文の削除や変数名の変換等を行う．その後，各メソッドからハッシュ値を計算し，コードクローンの検出を行う．また，石原らは 13,000 以上ものプロジェクトからなる UCI source code data sets[3][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 1,160,000 ものメソッド単位のコードクローンを検出した．さらに，検出されたコードクローンの内，多くのプロジェクトに含まれている 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．

4. 研究の動機

石原らの手法を用いることでライブラリ化を行う場合に有用なコードクローンを検出することができる．しかし，石原らの手法では，プログラムの機能が同じであっても異常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．異常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 1 に示す．この 2 つのメソッドの違いは，左のメソッドにおいて 6〜8 行目に存在している try〜catch 文の有無のみであり，異常処理は異なるがプログラムの機能は等しい．そのため，ライブラリ化を行う候補を検出する際には，この 2 つのメソッドを 1 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，この 2 つ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，異常処理に関するコードを除去したうえでメソッド単位のコードクローン検出を行う．

5. 異常処理を考慮したメソッド単位のコードクローン検出

5.1 概要

本研究では提案手法を石原らの手法に処理を追加することで実現する．そのため提案手法の入出力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．石原らの手法の概要を図 2 に示す．本研究では，

石原らの手法の正規化処理に対し異常処理の除去を追加する．また，本研究では Java で記述されたプログラムを対象としている．

5.2 各 STEP の説明

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．本研究では，ソースファイルから抽象構文木を作成することでメソッドの切り出しを実現する．抽象構文木を作成することで，改行や空白回数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に対して正規化を行う．行う正規化は，変数名，文字列リテラル，メソッド宣言部のメソッド名の特殊文字列への置換，および，修飾子，アノテーション，コメント文の削除である．本研究ではこの正規化処理を行う際に異常処理の除去を行っている

STEP3：**フィルタリング** ソースファイルには処理が単純であり，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する．

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 1 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．等しい記述を持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．本研究では，MD5[13] を用いてハッシュ値を計算し，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持つメソッドは等しい記述を持つ．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 1 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．

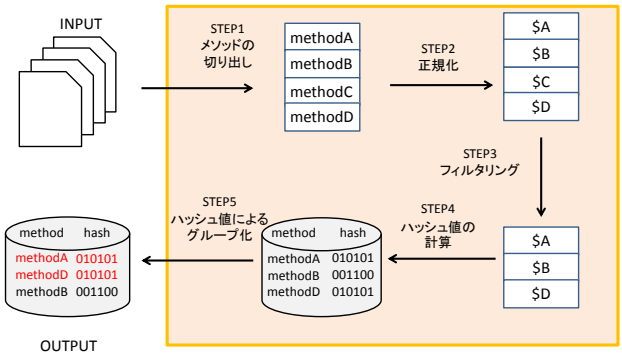

図 2. 石原らの手法の概要

5.3. 異常処理の除去

各メソッドから異常処理を検出し，除去を行う．本研究では 2.1 節に示した異常処理を対象としており，Java においては以下に示す処理が除去の対象となる．

- throw 文
- try〜catch〜finally 文と catch 節内部の処理
- throws 節
- 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
- System.exit() メソッド

異常処理の除去では，まず，切り出された各メソッドのソースコードを探索し，異常処理を検出する．次に，検出された異常処理をソースコードから除去する．最後に異常処理の除去によって空になった複文の除去を行う．

図 3，4 に異常処理除去の例を示す．図 3 では，throw 文と throws 節を除去している．この例では，まず除去前のコードから 4 行目の throw 文と 2 行目の throws 節が除去される．その後，4 行目の throw 文の除去により，

```
1 public static byte[] OR(byte[] a,byte[] b)
2    throws Exception{
3    if(a.length != b.length) {
4       throw new Exception();
5    }
6    byte[] v = new byte[a.length];
7       for(int i=0; i<v.length; i++) {
8       int A=(a[i]<0 ? a[i]+256 : a[i]);
9       int B=(b[i]<0 ? b[i]+256 : b[i]);
10      v[i]=(byte)(B | A);
11   }
12   return(v);
13}
```

除去前のコード

```
1 public static byte[] OR(byte[] a,byte[] b)
2  {
3    byte[] v = new byte[a.length];
4    for(int i=0; i<v.length; i++) {
5       int A=(a[i]<0 ? a[i]+256 : a[i]);
6       int B=(b[i]<0 ? b[i]+256 : b[i]);
7       v[i]=(byte)(B | A);
8    }
9    return(v);
10}
```

除去後のコード

図 3. throw 文，throws 節の除去例

```
1 public String read(){
2    readLock.lock();
3    String data=null;
4    try {
5       while(hasNext()){
6          data= new String(doRead());
7       }
8    } catch(Exception e){
9       e.printStackTrace();
10      data=null;
11   } finally {
12      readLock.unlock();
13   }
14   return data;
15}
```

除去前のコード

```
1 public String read(){
2    readLock.lock();
3    String data=null;
4    while(hasNext()){
5       data= new String(doRead());
6    }
7    readLock.unlock();
8    return data;
9 }
```

除去後のコード

図 4. try～catch～finally 文の除去例

3～5 行目の if 文が空となるために除去され，除去後のコードとなる．

図 4 では，try 文の除去を行っている．この例では，除去前のコードから 4～13 行目の try～catch～finally 文が除去される．さらに，catch 節の内部に存在する処理は異常が発生した場合にのみ実行されるため，異常処理として除去を行う．

5.4. 実装

本研究では，石原らの作成したメソッド単位のコードクローン検出ツールを改良し，提案手法の試作を行った．試作したツールの解析対象言語は Java のみであり，抽象構文木の作成には JavaDevelopmentTools[14] を用いている．また，石原らの手法と比較を行うために，正規化やハッシュ値の計算に関する処理は変更していない．そのため，ハッシュ値の計算には MD5 を用いており，行う正規化は以下の 2 つである．

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

また，空白や改行回数に関しては抽象構文木を用いることで，その違いを吸収している．

6. 実験

6.1 実験の準備

本研究では，提案した異常処理の除去を追加したコードクローン検出の有用性を示すために以下の 2 つの実験を行った

実験 1 異常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を調査する

実験 2 新たに検出されたコードクローンがライブラリ化に利用可能か調査する

実験 1 では，同一の対象に対して石原らの作成したメソッド単位のコードクローン検出ツールと，提案する異常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ツールを適用する．この際，追加した除去処理の影響のみを調査するため，各ツールのフィルタリング処理を取り除いた状態で実験を行った．これは，本研究では異常処理の除去を行う際に try 文や空になった複文を削除しており，石原らの手法におけるフィルタリング条件である複文の数に異常処理の除去が影響を与えてしまうためである．

実験 2 では，検出されたコードクローンがライブラリ化に有用であるかを，著者の 1 人が実際に目で見て調査した．また，実験 2 でコードクローン検出を行う際には，各メソッドのトークン数を用いてフィルタリングを行っている．これは，多くのソフトウェアに存在するがライブラリ化には不向きである，処理が単純で短いメソッドを検出しないためである．

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”[3][12] から Java 以外のファイルを削除したデータセットを使用した．さらに，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．本実験における実験対象の構成を表 1 に示す．本実験は 2CPU,8core(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．

本実験おいて，クローンセットに含まれる各メソッド内で参照されるフィールド数の最大値を NFR(the Number of Field References) と定義し，使用する．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．フィールドの参照が多いメソッドは外部への依存が強く，ライブラリ化を行うことが難しい．このようなメソッドを含むをコードクローンは，検出された際に NFR の値が大きくなる．したがって，NFR の値が小さいコードクローンほど，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. 実験 1 異常処理の除去による影響調査

実験対象に対し，試作ツールと石原らのツールを適用した結果を表 2 に示す．試作ツールの結果と石原らのツールとを比較すると, 検出したクローンセット数は減少しているが，クローンセットに含まれるメソッド数は増加している．これは，プログラムの機能が同じであるが異常処理が異なるために，コードクローンとして検出されていなかったメソッドや異なるクローンセットとして検出されていたメソッド群を，1 つのクローンセットとして検出できるようになったためであると考えられる．

そこで，異常処理の除去によりハッシュ値が変化したメソッドや，要素が変化したクローンセットがどの程度存在するか調査した．その結果を表 3，4 に示す．

この結果より，約 24%のメソッドが何らかの異常処理を行っていることがわかる．また，異常処理の除去により，プログラムの機能が同じであるが異常処理が異なるメソッド群を，1 つのクローンセットとして検出できるようになったことがわかる．

表 1. 実験対象の構成

全ファイル数	3,963,896
検出対象ファイル数	2,072,490
ソフトウェア数	13,193
検出対象メソッド数	19,416,603
検出対象ファイルの総行数	361,663,992
全容量	30.6GByte

6.3 実験 2 ライブラリ化に利用できるかの調査

実験 2 では，検出されたコードクローンがライブラリ化に有用であるかの調査を行う．ここで，処理が単純であるためにライブラリ化に適さないメソッドを検出しないように，メソッドに含まれるトークン数が 50 以上のメソッドのみを検出対象とした．しかし，検出されるコードクローンの数が膨大になるため，実験 2 では全てのコードクローンを分析するのではなく，石原らの手法で用いられている NOP メトリクス値 (コードクローンに含まれるメソッドが分布しているプロジェクト数) の上位 100 位 のコードクローンを実際に目で見て調査した．また，NFR 値が 0 のコードクローンに対しても，同様に NOP 値の上位 100 位について調査を行った．ここで，両方の調査対象に重複があるため，実際に調査したコードクローンは 179 である．

本実験では，調査したコードクローンの内，133 をライブラリ化すべきであると判断した．このうち，NFR 値が 0 であるコードクロ―ンは 81 存在する．

図 5 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 2 つのメソッドは，ともに 2 つの配列の連結を行っており，try〜catch 文の有無のみが異なっている．このメソッドを含むコードクローンは，40 のプロジェクトにまたがって存在する．このような処理は，今後も多くのプロジェクトで行われる可能性があり，かつ，フィールドへの参照が存在しないため，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 6 に示す．この 2 つのメソッドはユーザ定義名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プロジェクトに存在するが，処理が単純であり，かつ，フィールドへの参照が多いためライブラリ化を行う候補として検出すべきでない．しかし，メソッドに含まれるトークン数が大きいために，フィルタリングを通過して検出されてしまっている．

6.4. 実行時間の比較

本実験では，試作したツールを対象のデータセットに提要した結果，6.37 時間でコードクローンの検出が完了した．また，石原らの手法を実装したツールを用いた場合は，実行に 5.31 時間を要した．表 5 に各処理に要した時間を示す．

表 4. 異常処理除去によるクローンセットへの影響

新たに検出されたクローンセット数	38,206
複数のクローンセットが統合したクローンセット数	77,553
消失したクローンセット数	6

表 5. 解析時間

処理内容	試作ツール	石原らのツール
メソッドの切り出し 正規化 ハッシュ値の計算	4.66h	3.67h
ハッシュ値のグループ化	1.71h	1.64h
合計	6.37h	5.31h

表 2. 実行結果

	試作ツール	石原らのツール
検出したクローンセット数	1,895,289	1,953,211
クローンセットに含まれるメソッド数	13,883,691	13,670,771

表 3. 異常処理除去によるメソッドへの影響

ハッシュ値が変化したメソッド数	4,623,922(23.8%)
ハッシュ値が変化しなかったメソッド数	14,792,681(76.2%)

```
1 private static byte[] concatBytes(
          byte[] array1, byte[] array2) {
2    byte[] cBytes =
               new byte[array1.length
                 + array2.length];
3    try {
4       System.arraycopy(array1, 0, cBytes,
5               0, array1.length);
6       System.arraycopy(array2, 0,cBytes,
7               array1.length, array2.length);
8    } catch (Exception e) {
9       throw new RuntimeException(e);
10   }
11   return cBytes;
12}
```

```
1 private static byte[] concatBytes(
          byte[] array1, byte[] array2) {
2    byte[] cBytes =
               new byte[array1.length
                 + array2.length];
3    System.arraycopy(array1, 0, cBytes,
            0, array1.length);
4    System.arraycopy(array2, 0,cBytes,
            array1.length, array2.length);
5    return cBytes;
6 }
```

図 5. ライブラリ化すべきと判断したコードクローン

```
1 public int hashCode(){
2    final int prime = 31;
3    int result = 1;
4    result = prime * result +
       ((id == null)?0 :id.hashCode());
5    result = prime * result +
       ((key == null)?0 :key.hashCode());
6    result = prime * result +
       ((value == null)?0 :value.hashCode());
7    return result;
8 }
```

```
1 public int hashCode() {
2    final int prime = 31;
3    int result = 1;
4    result = prime * result +
       ((description == null) ?0
              : description.hashCode());
5    result = prime * result +
       ((items == null) ? 0 : items.hashCode());
6    result = prime * result +
       ((name == null) ? 0 : name.hashCode());
7    return result;
8 }
```

図 6. ライブラリ化すべきでないと判断したコードクローン

7. 考察

7.1. 新たに検出されたコードクローンの有用性

実験 2 で調査を行った結果，多くのプロジェクトに存在するコードクローンの内，70%以上のコードクローンがライブラリ化に有益であった．さらに，NFR 値が 0 であるコードクローンを対象にした調査では，検出したコードクローンの内 80%以上がライブラリ化に有益であった．また，石原らの手法において NOP メトリクス値の上位に存在していた size や close 等のライブラリ化には適さないメソッドを含むコードクローンが，実験 2 の調査では検出されなかった．これは，このようなメソッドで頻繁に用いられる特定の変数が null や 0 であれば例外を発生させるといった異常処理が除去されたことにより，メソッドサイズが小さくなり，フィルタリングを通過しなくなったためである．

7.2. 実行時間の変化に関して

試作したツールと石原らのツールの実行時間を比較すると，試作したツールでは正規化処理に時間がかかっている．これは，正規化処理に対し，異常処理の除去を追加しているためである．また，ハッシュ値のグループ化に要する時間はわずかに増加しているが，これは表 2 で示したようにクローンセットに含まれるメソッド数が増加したためである．しかし，試作したツールでは，360,000,000 行のソースコードに対し 6.37 時間で検出を終えており，実用的な時間で実行可能であるといえる．

8. 妥当性への脅威

8.1. 異常処理の定義について

本研究では，例外処理やロギング処理を異常処理として除去の対象としている．しかし，異常処理とみなす処理

を変更することによって本研究で得られた結果と異なる結果が得られる可能性がある．

8.2. ハッシュ値の衝突に関して

提案手法では 2 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異なるメソッドの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．ただし，多くのソフトウェアに適用していけば，いずれハッシュ値の衝突が起こる可能性がある．

8.3　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元にしており，以下の正規化処理を行っている．

- 変数名，文字列リテラル，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．

また本研究の実験 1(6.1 節) では異常処理を除去したことによる影響を調べるため，フィルタリング処理を行わずに評価している．しかし，型名の正規化の有無やメソッドサイズでのフィルタリング等の正規化やフィルタリング処理の方法を変更することによって，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．

9. おわりに

本論文では，異常処理のみが異なるメソッド群を 1 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，異常処理の除去をメソッド単位のコードクローン検出手法に追加し，検出を行った．今後の課題として，

- 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 (A)(課題番号：21240002)，萌芽研究 (課題番号：23650014，24650011)，若手研究 (A)(課題番号：24680002)，若手研究 (B)(課題番号:24700030) の助成を得た．

参考文献

[1] S.Livieri, Y.Higo, M.Matushita, and K.Inoue. Very-large scale code clone analysis and visualization of open source programs using distributed ccfinder: D-ccfinder. In *Proc. of the 29th International Conference on Software Engineering*, pp. 106–115, 2007.

[2] Y. Sasaki, T. Yamamoto, Y. Hayase, and K. Inoue. Finding file clones in freebsd ports collection. *Proc. of the 7th Working Conference on Mining Software Repositories*, pp. 102–105, May 2010.

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

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

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

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

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

[8] R. Koschke. Large-scale inter-system clone detection using suffix trees. *Proc. of the 16th European Conference on Software Maintenance and Reengineering*, pp. 309–318, Mar. 2012.

[9] T. Ishihara, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto. Inter-project functional clone detection toward building libraries - an empirical study on 13,000 projects -. *Proc. of the Working Conference on Reverse Engineering*, pp. 387–391, Oct. 2012.

[10] Suman Saha, Julia Lawall, and Gilles Muller. An approach to improving the structure of error-handling code in the linux kernel. *Proc. of the SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems*, pp. 41–50, April 2011.

[11] Chen Fu and Barbara G. Ryder. Navigating error recovery code in java applications. *Proc. of the OOPSLA workshop on Eclipse Technology eXchange*, pp. 40–44, Oct. 2005.

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

[13] R. Rivest. The md5 message-digest algorithm. *RFC 1321(Informational)*, Apr. 1992. `http://www.ietf.org/rfc/rfc1321.txt`.

[14] Java development tools. `http://www.eclipse.org/jdt/`.