Vol.2022-SE-210 No.28
2022/3/12

型を考慮したコードクローンの集約リファクタリング

杉原 裕太[1,a)]　沖野 健太郎[1,b)]　清水 一輝[1,c)]　近藤 将成[1,d)]　亀井 靖高[1,e)]　鵜林 尚靖[1,f)]

概要：ソースコード中に点在するコードクローンは，ソフトウェアを保守する際に悪影響をおよぼす恐れがあると考えられている．それらのコードクローンの検出は，メソッド抽出リファクタリングを行う上で重要なステップとなり，多くの手法が提案されてきた．ところが，検出されたコードクローンがリファクタリング対象として適切かどうかの判断は困難であり，クローン情報のリファクタリング利用への障壁となっている．そこで本稿では型付き構文木を用いることで，メソッド抽出リファクタリングを見据えたコードクローンを検出する手法を提案する．さらに，提案手法によって検出されたコードクローンのリファクタリングに対する応用について論じる．

キーワード：コードクローン，型付き構文木，リファクタリング

1. はじめに

ソフトウェア開発において，コードクローンの存在はソフトウェアの保守を難しくすると言われている [1]．そのため，コードクローンを特定しリファクタリングによって集約することは，ソフトウェア開発において重要な課題とされており，現在に至るまで数多くのコードクローン検出手法が提案されてきた [2][3][4][5]．

ところが既存手法では，検出されたコードクローンがリファクタリング対象として適切かどうか評価する手段に乏しいという問題がある．実際，先ほど挙げた研究では，検出されたコードクローンがリファクタリング対象として適切かどうかの評価を目視に頼っている．リファクタリング対象として適切なコードクローンの選定を行うには，ソースコードの振る舞いの一致に基づいた解析が必要である．

そこで本研究によって，コードクローンの振る舞い上の評価を行うため，コードクローン中の式の型や参照を明確にした解析手法を提案する．そして型や参照の情報を，リファクタリングの対象として適切なコードクローンの選定や，具体的な抽出リファクタリングの操作に用いる．

本研究では，今日広く使われているオブジェクト指向言語のひとつである Java を対象とする．Java を用いて書かれたソースコードに対して評価実験を行い，従来手法によって検出できるコードクローンとの違いについて論じる．

本稿では，2 節で背景と動機を述べる．3 節で本稿中にて使用される用語について述べ，4 節で手法の提案を行い，5 節で実験と考察について述べる．最後に 6 節で今後の課題，7 節でまとめを述べる．

2. 背景と動機

2.1 コードクローンの検出技術

肥後ら [6] は，コードクローンの検出技術について次のように分類している．

- 行単位の検出
- トークン単位の検出
- 抽象構文木を用いた検出
- プログラム依存グラフを用いた検出
- 上記以外の技術を用いた検出

また，コードクローンは文字列の一致度にしたがって，次の 3 つに分類される [7]．

タイプ 1：空白，タブ，コメントを除いて一致するコードクローン

タイプ 2：タイプ 1 のコードクローンに加えて，識別子，

[1] 九州大学
Kyushu University
[a)] sugihara@posl.ait.kyushu-u.ac.jp
[b)] okino@posl.ait.kyushu-u.ac.jp
[c)] shimizu@posl.ait.kyushu-u.ac.jp
[d)] kondo@ait.kyushu-u.ac.jp
[e)] kamei@ait.kyushu-u.ac.jp
[f)] ubayashi@ait.kyushu-u.ac.jp

リテラル，型などの相違をもつ構造的に一致するコードクローン

タイプ 3：タイプ 2 のコードクローンに加えて，文の変更，追加や削除を含むコードクローン

行単位の検出やトークン単位の検出では，タイプ 2 のクローンを検出可能である．字句単位の解析を行いコードクローンを検出するものに，CCFinder[2] などがある．

抽象構文木を用いた検出でも同じく，タイプ 2 のクローンを検出可能である．ただしこちらはプログラムの構造に基づいて分析を行うため，プログラムの構造を無視したコードクローンを検出対象から除くことができる．また、演算子によって結合された識別子を 1 つのトークンとして見なすなどの構造的な評価も可能になる．抽象構文木を用いてコードクローンを検出するものに，Deckard[3] などがある．

プログラム依存グラフを用いた検出では，タイプ 3 のコードクローンを検出可能である．プログラム依存グラフを用いてコードクローンを検出するものに，ReAF[4] などがある．本研究では，構文木に基づいた解析を行うため，タイプ 2 のコードクローンを対象とする．

2.2 タイプ 2 のコードクローンと関連研究

定義上では，タイプ 2 のコードクローンはあらゆる識別子の相違を許容することになっている．すなわち，演算子やセパレータなどの位置が同じならば，識別子の大半が異なるようなものもタイプ 2 のコードクローンに含まれる．Simon[8] は，このようなコードクローンはリファクタリング対象として不適切であるとし，タイプ 2 のクローン検出において識別子名の一致割合による閾値を設定した．ところが，識別子名の一致割合によるフィルタリングには 2 つの問題点がある．

(1) 識別子名が同じであるが，参照が異なるような識別子も同じものとして扱ってしまう．
(2) 識別子名が異なっていても，プログラム上の意味がほとんど同じ識別子を異なるものとして扱ってしまう．

これは，識別子名の一致が必ずしもプログラム上での参照の一致を意味するわけではないことに起因する．例として**図 1** を示す．図 1 において 2 つのコード片は完全に同じ文字列表現となっているが，これらは定義されているメソッドが異なる．それに応じて一部の識別子の示すローカル変数や引数の参照も異なるため，同じ識別子名でありながら型の違う識別子の組が存在する．たとえ文字列表現が同じでも，このようなコードクローンは

```
do {
        count1 = input1.read(array1, pos1, DEFAULT_BUFFER_SIZE - pos1);
} while (count1 == 0);
if (count1 == EOF) {
        return pos2 == index && input2.read() == EOF;
}
pos1 += count1;
```

input1, input2 … InputStream | array1 … byte[]

```
do {
        count1 = input1.read(array1, pos1, DEFAULT_BUFFER_SIZE - pos1);
} while (count1 == 0);
if (count1 == EOF) {
        return pos2 == index && input2.read() == EOF;
}
pos1 += count1;
```

input1, input2 … Reader | array1 … char[]

図 1 表現が完全に同じであるが，識別子の型が異なるコードクローンの例

集約が難しく，リファクタリング対象として適切ではない．後に Simon は (1) の問題の解決のため，識別子の定義の解析を併用した検出を行ったものの [5]，(2) の問題の解決には至っていない．

2.3 本研究の目的

図 1 のような例を含めて，識別子名の一致率をコードクローンのフィルタリングに用いることは，必ずしもリファクタリングに直結するコードクローンの選定につながるわけではない．よって本研究では構文木を用いて取得されたコードクローンへのフィルターとして，識別子名の一致率ではなく，型や参照の差異を用いる．そのために，識別子を表すノードそれぞれに参照を持たせ，それに併せて式を表すノードに型付けが行われた具象構文木をソースコードから構築する．そして構造が等しい構文木の各ノードの差異を用いて，リファクタリングに適したコードクローンの選定，およびリファクタリングに必要な情報の取得を達成する．

3. Java における型の取り扱い

手法の説明や考察の際に用いる Java の型の取り扱いについて述べる．これらは，Java のリファレンス [9] に示されているものを簡潔な説明に書き直したものである．

3.1 型の互換

型 S の値が型 T の変数として代入可能であるとき，型 S は型 T と互換であるという．これを $S \to T$ と表す．

3.2 サブタイプとスーパータイプ

$S \to T$ であり，S と T がともにプリミティブ型，あるいはともに参照型か交差型であるとき，型 S は型 T のサブタイプであるといい，これを $S <: T$ と表す．逆

に，型 S が型 T のサブタイプであるとき，型 T は型 S のスーパータイプであるという．

3.3 型の等価

型 S が型 T と等価であるとき，これを $S = T$ と表す．

3.4 最小上界

型 $U_1, ..., U_n$ の共通のスーパータイプの集合を A とする．A に属する 1 つの型 T について，A に属する他の全ての型が T のスーパータイプであるとき，T を $U_1, ..., U_n$ の最小上界 (least upper bound) といい，$lub(U_1, ..., U_n)$ と表す．

3.5 型推論

型推論とは，型変数をもったメソッドやコンストラクタを型引数を明示せずに参照したり，コンストラクタにダイヤモンド演算子（<>）を用いたりした場合に，明示されなかった型を引数などから推論する仕組みである．型引数の代わりに推論変数を型変数に代入し，引数の互換性や返り値型の互換性から推論変数に型の制約を付け，型引数を決定する．

4. 型を考慮したコードクローン検出手法

本研究では，まずソースコードの構造的な比較を行いやすくするため，Java のソースコードを構文木（4.1 節）に変換する．ソースコードの振る舞いに基づいた解析のため，この構文木は従来の抽象構文木と次のような点で異なる．

（1）識別子を表すノードに識別子の参照が保持される．
（2）式を表すノードに型の参照が保持される．式ノードの型は子ノードの型から動的に決定される（4.3 節）．
（3）ブロック中の各文が可換実行セット（4.2 節で後述）によって保持される．

（1）（2）は，識別子の参照や式の型をコードクローン検索の際に比較できるようにすることを目的としている．（3）は，どのような実行順序でも結果に影響しない文の組み合わせの，ソースコード中の記述順序を無視することで，より広い範囲でコードクローンを探せるようにすることを目的としている．

こうして構築された構文木は一定の手順でペアリングされ，そのうち構造的に一致しているものがコードクローンとして保持される (4.4 節)．コードクローンは式の型の差異，式の参照の差異，構文木サイズといった情報をもち，これらに制約をかけたフィルタリングが可能である．

4.1 構文木のノード

構文木のノードは，ブロックノード，文ノード，式ノードの 3 種類に大別される．

4.1.1 ブロックノード

Java プログラムにおけるブロックとは，複数の文を波括弧で囲んで記述したものである．ブロックノードは，ブロックに対応するノードであり，複数の文ノードおよび式ノードを子に持つ．ブロックノード中では，ソースコードで記述された文の実行順序の代わりに，可換実行セットによって文の実行順序を保持する．

4.1.2 文ノード

Java プログラムにおける文とは，プログラムの実行系列を制御する表現である．構造の一部として他の式，文，およびブロックを持つことがある．文ノードは文に対応するノードであり，元となった文の示す意味に応じて，他のノードを子に持つ．文ノードの種類を，**表 1** に示す．

4.1.3 式ノード

Java プログラムにおける式とは，void（返り値なし），あるいは何らかの値を返す表現である．すべての式はコンパイル時に型をもつ．式ノードは式に対応するノードであり，式と同じく明示的な型を持つ．式ノードの種類を，**表 2** に示す．

4.2 可換実行セット

可換実行セットとは，ブロックノードの内部に存在する構造で，どのような順番で実行してもプログラムの結果が変化しない文ノードの組み合わせである．一つのブロックノードは可換実行セットの組み合わせで構成され，文の実行順序に対応して可換実行セットにも順序が付けられている．可換実行セットの構築は，ノードのデータ依存関係と制御関係を考慮して行う．

4.2.1 可換でない文ノード

次のいずれかの条件を満たす 2 つの文ノードは，可換でない文ノードであり，同じ可換実行セットに加えることができない．

（1）どちらか一方, あるいは両方が return ノード, throw ノードを子に含む．
（2）どちらか一方，あるいは両方が現在のブロックの実行を中断する break ノード，continue ノードを子に含む．
（3）両方がそれぞれメソッドノードを子に含む．

（4）一方がフィールドの更新を子に含み，もう一方がメソッドノードを子に含む．
（5）一方がフィールドの参照を子に含み，もう一方がメソッドノードを子に含む．
（6）両方が同じフィールドの更新を子に含む．
（7）両方が同じローカル変数の更新を子に含む．
（8）片方がフィールドの参照を子に含み，もう片方が同じフィールドの更新を子に含む．
（9）片方がローカル変数の参照を子に含み，もう片方が同じローカル変数の更新を子に含む．

4.2.2 ブロックノードの構築

ブロック中の文は，ソースコードの記述順に読み取って文ノードに変換する．もし読み取った文が switch 文の case ラベルの直後にあるならば，ブロックノードに新たな可換実行セットを追加し，そこへ文ノードを case ラベルの値とともに登録する．switch 文の case ラベルの直後でない場合は，既にブロックノード中に存在している可換実行セットを実行順の逆から参照していく．もし参照中の可換実行セットに読み取られた文ノードと可換でない文ノードが存在するなら，文ノードをその後続の可換実行セットに登録する．後続の可換実行セットがない場合は，ブロックノードに新たな可換実行セットを追加し，そこへ文ノードを登録する．

4.3 式ノードの型の決定

全ての式ノードは型を持つが，式ノード自身が型の情報を持つ場合と，子ノードの型から式ノードの型が決定される場合がある．下記の式ノードは，子ノードの型に基づいて自身の型が決定される．

- 配列アクセスノード
- 返り値型の決定に型推論を必要とするメソッドノード
- 生成型の決定に型推論を必要とするコンストラクタノード
- 論理演算子以外の演算子ノード

4.4 コードクローンの検索

コードクローンの検索はクラス単位で行う．クラス内の各メソッドについて，メソッド定義ブロックノード直下の子ノードを取り出し，それらから構文木ペアを構成し比較していく．

4.4.1 クローンとなる構文木

2 つの構文木の比較は，根ノードから順番に行う．次の操作でクローンとなったペアのことを，以降クローン構文木と呼ぶ．

表 1 文ノードの種類

ノード名	構成要素
assert	式ノード（boolean），式ノード（String）
break	ラベル
continue	ラベル
return	式ノード
throw	式ノード（Throwable）
if-then-else	式ノード（boolean），ブロックノード/文ノード
switch	式ノード，ブロックノード
while	式ノード（boolean），ブロックノード/文ノード
do-while	式ノード（boolean），ブロックノード/文ノード
for	変数宣言/式ノード，式ノード（boolean），式ノード，ブロックノード/文ノード
foreach	変数宣言，式ノード（Iterable），ブロックノード/文ノード
synchronized	宣言/式ノード，ブロックノード
try	式ノード（Closeable），変数宣言，ブロックノード

表 2 式ノードの種類

ノード名	構成要素	型
配列アクセスノード	式ノード（配列型） 式ノード（数値）	配列型式の要素型
配列生成ノード	生成型 式ノード（数値）	生成型
配列初期化子ノード	代入先の型 式ノード	代入先の型
変数ノード	変数への参照	参照先の型
メソッドノード	メソッドへの参照 型引数 式ノード	推論された返り値型
コンストラクタノード	コンストラクタへの参照 生成型 コンストラクタ型引数 式ノード	推論された生成型
ラムダ式ノード	ラムダ式引数 ラムダ式の処理 代入先の型	代入先の型
メソッド参照ノード	メソッドへの参照 型引数 代入先の型	代入先の型
型キャストノード	キャスト型 式ノード	キャスト型
型トークンノード	型	型
リテラルノード	リテラル	リテラルの型
演算子ノード	演算子 式ノード	演算子の型

ブロックノード同士の場合．演算セットの数，及び各演算セットの持つ文ノードの数がすべて等しい場合，子ノードの比較に移る．異なる場合，クローンとしない．
文ノード同士の場合．文の種類が等しい場合，子ノードの比較に移る．異なる場合，クローンとしない．本調査は例外として，等価比較が難しい switch 文ノード同士の場合も，無条件でクローンとしない．
式ノード同士の場合．式ノード自体が持つ情報を比較し，同じなら子ノードの比較に移る．異なる場合，式ノードの差異を記録する．例外として，子ノードがともにノー

表 3　本手法で生成したクローン構文木の総数

構文木サイズ	総数	表現一致度		
		90%未満	80%未満	70%未満
6~10	1,034	899	597	424
11~15	315	194	49	28
16~20	28	20	10	0
21~25	5	3	1	0
26~30	1	0	0	0
31~35	1	0	0	0
36~40	2	0	0	0
41~	4	0	0	0
合計	1,390	1,116	657	452

ドの差異となるメンバアクセス演算子ノードは，それ自身をノードの差異として記録する．さらに本調査は，等価比較が難しいラムダ式ノードとメソッド参照ノードを，無条件にノードの差異として記録する．

違う種類のノードの場合．ブロックノードと文ノード，式ノードと文ノードなど，ノードの種類自体が異なる場合はクローンとしない．

4.4.2　式ノードの差異

得られた式ノードの差異は，参照の差異や型の差異としてクローン構文木に保持する．一方の式ノードを e_l，もう一方の式ノードを e_r，e_l の型を T_l，e_r の型を T_r とおく．式ノードの差異を参照の差異 $< e_l \Leftrightarrow e_r >$ とし，クローン構文木に記録する．さらに，$T_l = T_r$ でなければ，型の差異 $< T_l \Leftrightarrow T_r >$ をクローン構文木に記録する．なお，1 組のクローン構文木に同種のノードの差異が複数ある場合は，それらすべてを 1 個の差異として数える．

5.　実験結果と考察

5.1　実験の設定

対象となるソースコードは Java11 を用いて書かれており，コンパイル可能である必要がある．上記を満たすプロジェクトとして，本調査では 2022 年 1 月 13 日時点の Apache Commons IO[*1]を選定した．Apache Commons IO とは，JavaAPI の java.io パッケージの機能を拡張するオープンソースのライブラリである．このプロジェクトに対し，ディレクトリ src/main/java 以下の 222 のクラスおよびインタフェース直下の計 2,089 個のメソッドを解析した．

本調査は，4 節で挙げた式の型と参照に基づいた検出によって，従来の識別子の相違をもとにした解析では得ることができない恩恵について説明することを目的とする．そのため，2.2 節で述べた問題点をもとに次の 2 つについて検証する．

（1）文字列表現が一致あるいは類似しているが，リファクタリングに不適なコードクローンをどれだけ除くことが可能であるか．

（2）文字列表現に一定の相違があるが，リファクタリングに適したコードクローンをどれだけ検出することが可能であるか．

そこで本調査では，型の差異が存在せず，参照の差異が 2 個以下のクローン構文木をリファクタリングに適したコードクローンと設定する．これらのコードクローンは，引数が 2 個以下のメソッドに抽出できる可能性がある（5.3 節で後述）．さらに，識別子名の一致率によるコードクローン検出を想定し比較を行うため，クローン構文木に表現一致度という指標を導入する．クローン構文木を構成する全てのノードの数を n_A，そのうち文字列表現が異なるノードの数を dif とおくと，表現一致度は dif/n_A となる．ただし，参照の差異となるノードはその子ノードの数に関わらず 1 つのノードとして数えられる．本調査ではこの表現一致度に関して，90%未満，80%未満，70%未満の 3 つのフィルターを設けた．

[*1] https://github.com/apache/commons-io

表 4　型の差異が無く，参照の差異が 2 個以下のクローン構文木の総数

構文木サイズ	総数	表現一致度		
		90%未満	80%未満	70%未満
6~10	330	212	80	51
11~15	93	55	0	0
16~20	5	2	1	0
21~25	1	1	1	0
26~30	0	0	0	0
31~35	0	0	0	0
36~40	1	0	0	0
41~	0	0	0	0
合計	430	270	82	51

5.2　Apache Commons IO の解析結果と考察

Apache Commons IO に含まれる 2,089 個のメソッドのうち，1,976 個の解析に成功した．解析に失敗したメソッドについては，6.4 節で述べる．**表 3** に，検出できたクローン構文木数を示す．また，表 3 のクローン構文木のうち，型の差異が無く，参照の差異が 2 個以下だったものを**表 4** に示す．本調査では，5.1 節の（1）における文字列表現が一致あるいは類似するコードクローンと，5.1 節の（2）における文字列表現に一定の相違があるコードクローンを分ける際，2 つをおよそ半数ずつに分けることができる表現一致度 80%の閾値に着目する．

5.1 節の（1）に関して検証を行う．検出したクローン

構文木の総数のうち，表現一致度が 80%以上のクローン構文木は，表 3 から算出すると 733 組となる．また，型の差異が無く，参照の差異が 2 個以下のクローン構文木のうち，表現一致度が 80%以上のクローン構文木は，表 4 から算出すると 348 組となる．よって，型と参照の差異の制約を用いた検出によって，表現一致度が 80%以上のクローン構文木において 385 組 (52.5%) のクローン構文木をリファクタリングに不適当であるとして除くことができた．

次に 5.1 節の（2）に関して検証を行う．表 3 から，表現一致度が 80%未満のクローン構文木の総数は 657 組であり．表 4 から，そのうち 82 組（12.5%）がリファクタリングに適したコードクローンであった．よって，型と参照の差異の制約を用いた検出によって，表現一致度が 80%未満のクローン構文木において 12.5%のクローン構文木をリファクタリングに適当であるとして検出できた．

以上より，コードクローンの検出にかけるフィルターとして，型や参照の差異を用いることは有効であると考えられる．

5.3 検出したコードクローンのリファクタリング利用

従来の手法と異なり，本手法では全ての識別子の型と参照を明確にする．よって，クローン構文木とその差異を用いた以下のような検証と操作によって，メソッド抽出リファクタリングを行うことが可能となる．

（1）クローン構文木に対する制御的検証 (5.3.1 節)
（2）抽出メソッドの返り値の決定 (5.3.2 節)
（3）ノードの差異の解決 (5.3.3 節，5.3.4 節)
（4）抽出メソッドの修飾子と例外の決定 (5.3.5 節)

5.3.1 クローン構文木に対する制御的検証

クローン構文木において抽出リファクタリングが可能となる制御上の条件は，Emerson らの議論 [10] に基づいて考えると，以下のようになる．

（i）クローン内部で値が変更されるローカル変数のうち，クローンの後続の処理で参照されるものが 1 個以下である．そのローカル変数の更新値がメソッドの返り値となる．
（ii）クローン構文木中に return ノードが含まれている場合は，return によってコードクローン中の制御フローがすべて閉ざされている．
（iii）クローン構文木の外部のブロックを制御対象とする break ノード，continue ノードが存在しない．

（i）（ii）（iii）のいずれかの条件を満たさないクローン構文木は，抽出不可能である．

5.3.2 抽出メソッドの返り値の決定

抽出メソッドの返り値を，以下のように決定する．

- return によってクローン中の制御フローが閉ざされている場合，クローン内部で return によって返される値．
- クローン中に return が含まれていない場合，5.3.1 節の（i）におけるローカル変数の更新値．

返り値として決定した値に型の差異がない場合，値の型がそのままメソッドの返り値型となる．型の差異がある場合，5.3.3 節で決定された型がメソッドの返り値型となる．

5.3.3 型の差異の解決

型の差異が配列生成ノード，および instanceof 演算子ノードの子である型トークンノードに起因するものである場合，型の差異は解決できない．また，型の差異が throw ノードによって投げられる例外の差異に起因するものである場合も，抽出メソッドの投げる例外をその呼び出し元メソッドが処理できなくなるため，型の差異は解決できない．上記以外のとき，クローン構文木に保持された型の差異を $< T_l \Leftrightarrow T_r >$ とする。また，$< T_l \Leftrightarrow T_r >$ の型の差異をもつ参照の差異のうち 1 つを $< n_l \Leftrightarrow n_r >$ とおく．

n_l を左辺の子ノードとして持つメンバアクセス演算子ノードを列挙し，その右辺の子ノードを $e_{l1}, ..., e_{lk}$ とする．n_r に対しても同様にして，$e_{r1}, ..., e_{rk}$ を定義する．すべての $i = 1, ..., k$ に対して，次のいずれかの条件が成り立つとする．

(i) e_{li} と e_{ri} がともにメソッドノードで，$lub(T_l, T_r)$ またはそのいずれかのスーパータイプに属する同じメソッドを継承している．
(ii) e_{li} と e_{ri} がともに変数ノードで，$lub(T_l, T_r)$ またはそのいずれかのスーパータイプに属する同じフィールドを参照している．

このとき，参照の差異 $< n_l \Leftrightarrow n_r >$ は，抽出メソッドにおける 1 つの変数に集約することができる．同様にして，$< T_l \Leftrightarrow T_r >$ を誘引するすべての参照の差異をそれぞれ 1 つの変数に集約することができる場合，型の差異 $< T_l \Leftrightarrow T_r >$ は解決できる．型の差異の解決ができない場合，クローン構文木は抽出不可能である．

解決できる型の差異 $< T_l \Leftrightarrow T_r >$ をもつ参照の差異に，抽出メソッドの返り値として返される値が含まれている場合，あるいは $< T_l \Leftrightarrow T_r >$ がクローン構文木中で使用される型の型引数の差異として出現する場合，抽出メソッドに $lub(T_l, T_r)$ を上限境界とした型変数 U

```
In java/org/apache/commons/io/comparator/SizeFileComparator.java
size: 24 | variable differences: 2 | type differences: 0
textual differences: 6
differences: [size1 <-> size2], [file1 <-> file2]
```

```
if (file1.isDirectory()) {
        size1 = sumDirectoryContents && file1.exists() ? FileUtils.sizeOfDirectory(file1) : 0;
} else {
        size1 = file1.length();
}
```

```
if (file2.isDirectory()) {
        size2 = sumDirectoryContents && file2.exists() ? FileUtils.sizeOfDirectory(file2) : 0;
} else {
        size2 = file2.length();
}
```

図 2　検出されたコードクローンとノードの差異の例

```
//TODO rename method (auto-generated)
private long newMethod(final File p1) {
        if (p1.isDirectory()) {
                return sumDirectoryContents && p1.exists() ? FileUtils.sizeOfDirectory(p1) : 0;
        } else {
                return p1.length();
        }
}
```

図 3　抽出メソッド例

を追加する．$< T_l \Leftrightarrow T_r >$ をもつ参照の差異すべては，U を型とする変数として集約できる．そうでなければ，$< T_l \Leftrightarrow T_r >$ をもつ参照の差異すべては，$lub(T_l, T_r)$ を型とする変数として集約できる．

5.3.4　参照の差異の解決

参照の差異の解決は，以下のように行われる．

（i）差異となる式ノードがどちらもクローン以前の処理で参照されることがないローカル変数の場合，参照の差異は抽出メソッド中の新たなローカル変数になる．

（ii）差異となる式ノードが void メソッドの場合，参照の差異の解決はできない．（関数型インタフェースを用いれば厳密には可能であるが，本稿では対象外とする．）

（iii）差異となる式ノードが（i）（ii）のいずれにも該当しない場合，参照の差異は抽出メソッドの引数になる．

参照の差異の解決ができない場合，クローン構文木は抽出不可能である．

5.3.5　抽出メソッドの修飾子と例外の決定

クローン構文木の定義元メソッドの strictfp 修飾子，synchronized 修飾子の有無が一致していない場合，リファクタリング前後でプログラムの実行結果が等しいことが保証されないため，クローン構文木は抽出不可能である．上記以外のときクローン構文木は抽出可能で，アクセス修飾子以外の修飾子はクローン構文木の定義元メソッドから抽出メソッドに継承される．抽出メソッドの投げる例外は，クローン構文木中で投げられる例外のうち，try ノードでキャッチされないものである．

```
@Override
public int compare(final File file1, final File file2) {
        final long size1;
        if (file1.isDirectory()) {
                size1 = sumDirectoryContents && file1.exists() ? FileUtils.sizeOfDirectory(file1) : 0;
        } else {
                size1 = file1.length();
        }
        final long size2;
        if (file2.isDirectory()) {
                size2 = sumDirectoryContents && file2.exists() ? FileUtils.sizeOfDirectory(file2) : 0;
        } else {
                size2 = file2.length();
        }
        final long result = size1 - size2;
        if (result < 0) {
                return -1;
        }
        if (result > 0) {
                return 1;
        }
        return 0;
}
```

図 4　リファクタリング前のソースコード

```
@Override
public int compare(final File file1, final File file2) {
        final long size1 = newMethod(file1);
        final long size2 = newMethod(file2);
        final long result = size1 - size2;
        if (result < 0) {
                return -1;
        }
        if (result > 0) {
                return 1;
        }
        return 0;
}
```

図 5　リファクタリング後のソースコード

5.3.6　抽出リファクタリングの例

5.2 で実際に抽出したコードクローン（図 2）を例にとって，集約操作を行う場合を考える．まず，このコードクローンは 5.3.1 節の条件をすべて満たす．後続の処理で参照されるローカル変数 (size1, size2) が，集約先においてただ 1 つ存在するので，この値が抽出メソッドの返り値となる．また，ノードの差異はすべて解決可能であり，コードクローンの定義元メソッドは同じであるためメソッドの修飾子も同じである．よって，このコードクローンは新たなメソッドとして抽出できる（図 3）．元のソースコード（図 4）を抽出メソッドを用いて書き直すと，図 5 のようになる．

6.　今後の課題

6.1　データセットの量

本調査は Apache Commons IO の 1 プロジェクトの解析にとどまっている．そのため，コーディングスタイル等のソースコードの特徴に偏りがある可能性がある．より多くのプロジェクトに対しての検証が求められる．

6.2　構文木のペアリング

本手法では構文木のペアは，同じクラス内に定義され

たメソッドの，直下の文の間でのみ形成される．そのため，ブロック内部とブロック外部などのスコープの異なる構文木や，共通の継承先を持つなどの関連を持ったクラスやインタフェースの間での構文木のペアリングを行うことで，より多くのコードクローンを検出できる可能性がある．

6.3 バリエーション除去の不足

たとえ同じ処理を意味するプログラムであっても，その表現方法は多岐に渡り，これをバリエーションという [11]．バリエーションには次の 3 種類が存在する．

連接順序バリエーション：順番を入れ替えても実行結果が変化しない文の組み合わせのバリエーション．

構文バリエーション：if 文と switch 文，for 文と while 文など，処理を実現するために使用してある構文のバリエーション．

代入バリエーション：式の値を一旦別の変数に保持するか，あるいは保持せず式をそのまま使うかなど，値の代入関係のバリエーション．

コードクローンの検出の操作の前には，ソースコードはバリエーションをなるべく除去される形に変換されていることが望ましい．今回の手法では可換実行セットにより一部の連接順序バリエーションの解決を図っている．ところが構文バリエーション，代入バリエーションに関しては除去する操作を行っていない．これらのバリエーションを除去する操作を加えることで，より多くのコードクローンを検出できる可能性がある．

6.4 ツールの未対応事項

結果の項でも述べたが，ツールが対応していない構文の出現などの理由で，2089 のメソッドのうち 113 のメソッドの解析に失敗している．ツールが対応する構文の範囲を広げることで，より多くのコードクローンを検出できる可能性がある．

7. まとめと今後の展望

本調査では，型付き構文木を用いてコードクローン中の式の型や参照の差異を解析することにより，コードクローンの検出数への影響，および表現一致度による検出との違いについて検証した．その結果，表現一致度が 80%以上のクローン構文木全体において，52.5%のクローン構文木をリファクタリングに不適当であるとして除くことができた．また，表現一致度が 80%未満のクローン構文木全体において，12.5%のクローン構文木をリファクタリングに適当であるとして検出することができた．したがって，コードクローン中の式の型や参照を解析することは，よりソースコードの振る舞いに即したコードクローンの検出を可能にすると考えられる．それに加えて，コードクローン中の式の型や参照はメソッド抽出リファクタリングに活用できるため，有効性が高い手法であると結論付けられる．今後の展望として，6 節で挙げたような点に改善を施し，より効果的かつ広範な手法を確立することが挙げられる．

謝辞

本研究の一部は，JSPS 科研費 JP18H04097，JP21H04877，および，JSPS・スイスとの国際共同研究事業（JPJSJRP20191502）の助成を受けた．

参考文献

[1] 神谷年洋, 肥後芳樹, 吉田則裕. コードクローン検出技術の展開. コンピュータソフトウェア, Vol. 28, No. 3, 2011.

[2] Toshihiro Kamiya, Shinji Kusamoto, and Katsuro Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, Vol. 28, No. 7, 2002.

[3] Lingxian Jiang, Ghassan Misherghi, and Zhendong Su. Deckard: Scalable and accurate tree-based detection of code clones. *29th International Conference on Software Engineering*, 2007.

[4] 兼光智子, 肥後芳樹, 楠本真二. プログラム依存グラフを用いたリファクタリング候補の特定と可視化. 電子情報通信学会技術研究報告, Vol. 110, No. 336, pp. 61–66, 2007.

[5] Simon Baars. Statement-level ast-based clone detection in java using resolved symbols. In *Proceedings of the 18th Belgium-Netherlands Software Evolution Workshop, Brussels, Belgium, November 28th to 29th*, Vol. 2605 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2019.

[6] 肥後芳樹, 吉田則裕. コードクローンを対象としたリファクタリング. コンピュータソフトウェア, Vol. 28, No. 4, 2011.

[7] Chanchal Kumar Roy and James R. Cordy. A survey on software clone detection research. *School of Computing TR 2007-541, Queen's University*, Vol. 115, , 2007.

[8] Simon Baars. Improving software maintainability through automated refactoring of code clones. Master's thesis, University of Amsterdam, 2019.

[9] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. The java® language specification. 2021.

[10] Emerson Murphy-Hill and Andrew P. Black. Breaking the barriers to successful refactoring: Observations and tools for extract method. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, p. 421–430, New York, NY, USA, 2008. Association for Computing Machinery.

[11] 服部徳秀. プログラムの表現方法に関する研究. PhD thesis, Nagoya Institute of Technology, 1996.