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.

コードクローン検出ツールの差分情報を用いた不具合検出手法の提案と評価

澤　健一†　肥後　芳樹†　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　〒 560-8531 豊中市待兼山町 1-3
E-mail: †{k-sawa,higo,kusumoto}@ist.osaka-u.ac.jp

あらまし　近年，コードクローンに対する関心が高まってきている．コードクローンとは，主としてコピーアンドペースト等によってソースコード上に生じる類似したコード片のことで，コンパイラでは検出できないような潜在的な不具合を埋め込みやすいといわれている．コードクローンを検出する様々な手法があるが，各手法が用いているアルゴリズムが異なるため，検出されるコードクローンに差が生じる．本稿では，この差を利用して文の追加・削除や識別子の修正漏れのような，不具合の可能性が高いコードクローンのみを抽出する手法を提案し，その評価を行う．現在までに行った実験では，1 つのコードクローン検出ツールのみを用いた場合に比べて，非常に短時間で目的の不具合を発見できている．

キーワード　コードクローン，ソフトウェア保守，バグ検出

Proposal and evaluation of an approach to find bugs using difference information of code clone detection tools

Kenichi SAWA†, Yoshiki HIGO†, and Shinji KUSUMOTO†

† Graduate School of Information Science and Technology, Osaka University
1-3, Machikaneyama-cho, Toyonaka, Osaka, 560-8531, Japan
E-mail: †{k-sawa,higo,kusumoto}@ist.osaka-u.ac.jp

Abstract　Code clone has recently received a lot of attention. A code clone is a code fragment that has other similar or identical code fragments in the source code. Most code clones are generated by ‘copy and paste’ programming. Potential bugs, which cannot be detected by compiler, are often introduced due to code clones. At present, many code clone detection tools have been developed. However, different code clones are identified from the same source code by the tools since they adopt different detection algorithms. This paper proposes a method for efficiently detecting code clones that are very likely to include bugs. A software tool has been implemented based on the proposed method, and it was applied to LINUX source code. As a result, some bugged code clones were detected within a short time frame.

Key words　Code Clone, Software Maintenance, Bug Detection

1. は じ め に

近年，ソフトウェアの保守を困難にさせる要因として，コードクローンに対する関心が高まってきている．コードクローンとは，ソースコード上に存在する同一もしくは類似したコード片のことであり，コピーアンドペースト等によって発生する．あるコード片に不具合が存在した場合や，保守変更が必要になった場合，そのコード片とコードクローンになっている全てのコード片に対して同様の修正を行わなければならない．ソフトウェアの規模が拡大するほど修正に要する作業量が増大する．また，コピーアンドペーストを行った際に，コピー先に合わせて識別子を変更したり，新たな文を追加したりといった修正作業がしばしば行われる．この作業を慎重に行わないと，修正ミスや修正漏れといったことが発生する可能性がある．コードクローンにはこのような問題が存在するため，ソースコードからコードクローンを自動的に検出してその情報を用いることは，ソフトウェア開発や保守を行ううえで非常に有効である．

これまでに，コードクローンを検出する様々な手法が提案されているが，それらのツールはどれも異なるコードクローンの定義を用いているため，同じ対象からコードクローンを検出し

表 1　各検出手法によって検出されるタイプ

検出手法	検出されるタイプ				
	1a	1b	2a	2b	3
Ducasse の手法 [2]	○	○	×	×	×
Baker の手法 [3]	○	○	○	×	×
Kamiya の手法 [4]	○	○	○	○	×
Wettel の手法 [5]	○	○	○	○	○

た場合でもツールごとに検出結果が異なる．

本稿では，これら既存のコードクローン検出ツールにおける出力結果の差異に注目し，不具合の可能性が高いコードクローンのみを抽出する手法を提案する．この手法により，コードクローンに起因する不具合を効率よく特定することができる．

2.　コードクローンの分類と特徴

2.1　コードクローンの分類

Bellon らは，コードクローンを次の 3 つのタイプに分類している [1]．

タイプ 1：空白，改行位置など，コーディングスタイルのみが異なるコードクローン

タイプ 2：変数名，関数名や変数の型など，識別子のみが異なるコードクローン

タイプ 3：文の挿入，削除，変更が行われたコードクローン

本稿では，このうちタイプ 1，タイプ 2 をさらに分け，5 つのタイプに分類する．

タイプ 1a：表面上完全に同一なコードクローン

タイプ 1b：空白，改行位置に違いがあるコードクローン

タイプ 2a：パラメタライズドマッチング（P-Match）であるコードクローン（P-Match については次節で説明する）

タイプ 2b：P-Match でないコードクローン

タイプ 3：文の挿入，削除，変更が行われたコードクローン

表 1 に既存のコードクローン検出ツールが検出できるコードクローンのタイプをまとめる．各ツールによって検出することのできるコードクローンのタイプが異なっているのが分かる．

2.2　パラメタライズドマッチング（P-Match）

P-Match とは，ソースコードの比較を行うときに識別子の対応付けを行うものである．**図 1** に P-Match の例を示す．P-Match を用いている場合は，比較を行うときに識別子の違いを認識するので，**図 1** の code1 と code2 はコードクローンでないと判定される．一方，P-Match を用いない場合は識別子の違いを認識しないので，コードクローンであると判定される．

2.3　各タイプの特徴

各タイプに分類されたコードクローンはそれぞれ特徴が異なるため，起こりうる問題も異なる．以下の 3 つのタイプは，他のタイプに比べて問題が起こりやすいと考えられるものである．

タイプ 1b：プログラムの意味的には同一であるが，コーディングスタイルが異なるため，可読性が低い可能性が考えられる．

タイプ 2b：変数名，関数名や変数の型などの識別子が変更されているが，P-Match になっていないため，識別子の変更漏れ，もしくは変更し間違えている可能性が考えられる．

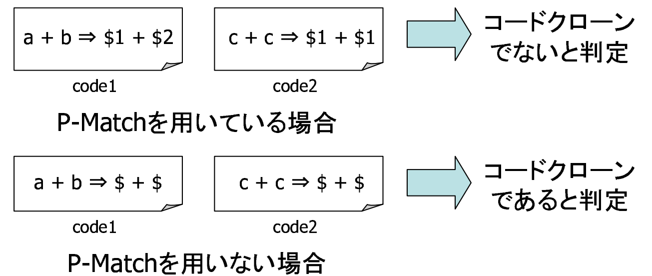

図 1　P-Match の例

タイプ 3：文の挿入，文の削除，文の変更が行われているため，文の挿入，削除，変更漏れの可能性が考えられる．

タイプ 1b のコードクローンは直接不具合の原因になる可能性は低いが，タイプ 2b，タイプ 3 のコードクローンは修正ミスが考えられるため，不具合の原因になる可能性が高い．本稿では，これらタイプ 2b，タイプ 3 のコードクローンに注目する．

図 2 は，提案手法により検出されたタイプ 2b，タイプ 3 それぞれにおける不具合の例を示している．**図 2(a)** では，コード 2 において変数名の修正漏れのため，不具合が引き起こされている．**図 2(b)** では，コード 2 において本来存在すべき必要な関数が無いため，不具合が引き起こされている．

3.　提 案 手 法

3.1　概　　要

表 1 で示したように，既存のコードクローン検出ツールは異なったタイプのコードクローンを検出する．それら異なるコードクローン検出ツールの差分を取ることで，特定のタイプのコードクローン情報のみを得ることができる．

例えば，Wettel の手法 [5] と Kamiya の手法 [4] の出力結果の差分を取ると，タイプ 3 のコードクローンを抽出できる．タイプ 3 のコードクローンは，文の挿入，削除，修正漏れの可能性が考えられるため，コード片間の異なる文に注目することで不具合を発見しやすくなる．また，Kamiya の手法と Baker の手法 [3] の出力結果の差分を取ると，タイプ 2b のコードクローンを抽出できる．タイプ 2b のコードクローンは，変数名，関数名や変数の型などの修正漏れの可能性が考えられるため，コード片間の識別子に注目することで不具合を発見しやすくなる．

提案手法はコードクローン検出ツールの出力結果を用い，直接対象ソースコードの解析は行わない．そのため，差分を取得する 2 つのコードクローン検出ツールが対応してさえいれば，どのようなプログラミング言語に対しても適用可能である．

2 つのコードクローン検出ツールにおける出力結果の差分を取得するには，それぞれのツールで検出されたコードクローンを比較し，同一であるかどうかを判断する必要がある．同一であると判断された場合，そのコードクローンを不要なものとして削除する．コードクローンの比較には，ソースコードのファイル名とコード片の開始行，終了行の情報を用いる．コードクローンの比較に行数を用いているのは，トークン情報などを持っていないコードクローン検出ツールが存在するためである．

コード1	コード2(不具合を含む)

File:linux-2.6.6/arch/m68k/mac/iop.c(line 246-257)
 	
if (macintosh_config->scc_type == MAC_SCC_IOP) {
    if (macintosh_config->ident == MAC_MODEL_IIFX) {
        iop_base[IOP_NUM_SCC] = (struct mac_iop *) SCC_IOP_BASE_IIFX;
    } else {
        iop_base[IOP_NUM_SCC] = (struct mac_iop *) SCC_IOP_BASE_QUADRA;
    }
    iop_base[***IOP_NUM_SCC***]->status_ctrl = 0x87;
    iop_scc_present = 1;
} else {
        iop_base[IOP_NUM_SCC] = NULL;
        iop_scc_present = 0;
    }
}
	File:linux-2.6.6/arch/m68k/mac/iop.c(line 258-269)
	
	if (macintosh_config->adb_type == MAC_ADB_IOP) {
	    if (macintosh_config->ident == MAC_MODEL_IIFX) {
	        iop_base[IOP_NUM_ISM] = (struct mac_iop *) ISM_IOP_BASE_IIFX;
	    } else {
	        iop_base[IOP_NUM_ISM] = (struct mac_iop *) ISM_IOP_BASE_QUADRA;
	    }
	    iop_base[***IOP_NUM_SCC***]->status_ctrl = 0; ***// 変数名の修正し忘れ***
	    iop_ism_present = 1;
	} else {
	    iop_base[IOP_NUM_ISM] = NULL;
	    iop_ism_present = 0;
	}

(a) タイプ 2b の不具合

コード1	コード2(不具合を含む)

File:linux-2.6.6/arch/alpha/kernel/time.c(line 531-539)
 	
int retval = 0;
int real_seconds, real_minutes, cmos_minutes;
unsigned char save_control, save_freq_select;
 	
/* irq are locally disabled here */
spin_lock(&rtc_lock);
 	
/* Tell the clock it's being set */
save_control = CMOS_READ(RTC_CONTROL);
CMOS_WRITE((save_control|RTC_SET), RTC_CONTROL);
	File:linux-2.6.6/arch/mips/dec/time.c(line 94-100)
	
	int retval = 0;
	int real_seconds, real_minutes, cmos_minutes;
	unsigned char save_control, save_freq_select;
	
	// 必要な関数が存在しない
	
	/* tell the clock it's being set */
	save_control = CMOS_READ(RTC_CONTROL);
	CMOS_WRITE((save_control | RTC_SET), RTC_CONTROL);

(b) タイプ 3 の不具合

図 2 コードクローンに起因する不具合の例

以下では，コードクローン比較の精度を上げるために行うソースコードの正規化と，コードクローンを同一であるかどうかを判断する 2 つの条件について述べる．

3.2 ソースコードの正規化

コードクローン検出ツールが用いている検出アルゴリズムの違いによって，ツールごとに検出されるコードクローンの開始位置，終了位置が若干異なる場合がある [1]．空行や中括弧のみである行の存在がその原因となっている．この問題を回避するため，次のようなソースコードの正規化を行う．

- 空行の削除
- コメント行の削除
- 不必要な空白，タブの削除
- 中括弧のみの行を削除し，1 つ上の行に中括弧を追加

ソースコードの正規化の例を**図 3** に示す．ソースコードの正規化を行うことで，コードクローン検出ツールにおけるコードクローンの開始位置，終了位置の違いを減らすことができるので，異なるコードクローン検出ツールで検出されたコードクローンであっても，次に述べる一致判定の精度が向上する．

3.3 コードクローンの一致条件 1

ここでは，コードクローンが同一であるかどうかを判断するための 1 つ目の条件を示す．

提案手法では，より差異のあるコードクローンを検出できるコードクローン検出ツールから，その差異を検出できないツールの差分を取ることで，タイプ 2b やタイプ 3 といった差異の大きなコードクローンを検出する．コードクローン検出ツールは，できるだけ大きな範囲をコードクローンとして検出しようとするため，より差異のあるコードクローンを検出可能なツールの方が，より大きな範囲をコードクローンとして検出する．したがって，一方のツールによる検出結果がもう一方のツールによる検出結果と完全に一致，もしくは内包されている場合のみ，それらのコードクローンが同一であると判断し，互い違いになっているような場合は同一であると判断しない．

```
while(c1){
    /* コメント */
    if(c2)
    {
        hoge();

    }
}
```

(a) 正規化前

```
while(c1){
if(c2){
hoge();}}
```

(b) 正規化後

図 3 正規化の例

図 4 は，2 つのコードクローンを CP_1，CP_2，それぞれのコード片を CF_1，CF_2 としたとき，2 つのコードクローンが互いに重なり合っている例である．この例の場合，$CP_1.CF_1$ と

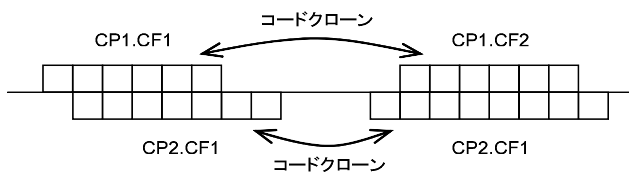

図 4　2 つのコードクローンが重なり合っている例

$CP_2.CF_1$ において，一方のコードクローンが内包されているという条件を満たさないため，CP_1 と CP_2 は異なるコードクローンであると判断される．

3.4　コードクローンの一致条件 2

次に，コードクローンが同一であるかどうかを判断するための 2 つ目の条件を示す．

2 つのコードクローン検出ツールによって検出されたコードクローンが完全に一致していれば問題は無いが，一方が内包されている場合は，どの程度の割合で一致しているかを判断する必要がある．この判断には good 値を用いる [1].

good 値を求める前準備として，2 つのコード片の一致割合を求める．2 つのコード片 CF_1，CF_2 の一致割合は，それぞれの行数を lines(CF_1)，lines(CF_2) と書くと，次の式で表される．

$$overlap(CF_1, CF_2) = \frac{|lines(CF_1) \cap lines(CF_2)|}{|lines(CF_1) \cup lines(CF_2)|}$$

この値を用いて，2 つのコードクローン CP_1，CP_2 の good 値 (p) は次のように表される．

$$good(CP_1, CP_2) = min\ (\ overlap(CP_1.CF_1, CP_2.CF_1), overlap(CP_1.CF_2, CP_2.CF_2))$$

$$good(CP_1, CP_2) \geqq p \qquad (p \in [0, 1])$$

2 つのコードクローンにおける good 値が p 以上を満たすとき，そのコードクローンが同一のものであると判断する．過去の研究において p の値は 0.7 が適切であると述べられており，本稿でもこの値を用いる [1], [3].

図 4 の例において good 値を求めると次のようになる．

$$good(CP_1, CP_2) = min(\frac{5}{8}, \frac{6}{8}) = \frac{5}{8} < 0.7$$

good 値が 0.7 よりも小さいため，**図 4** のコードクローンは異なるものであると判断される．ただし，good 値が 0.7 以上であっても，コードクローン一致条件 1 を満たさなければ同一であると判断されることはない．2 つの条件が同時に満たされて初めて，2 つのコードクローンが同一であると判断される．

4. 適用実験

4.1　概　　要

提案手法を実際のソフトウェアに適用することで，目的としているような不具合を抽出できるかどうかを調査する．対象ソフトウェアは Linux2.6.6/arch である．記述言語は C 言語，ファイル数 2,699 個，総行数 769,467 行となっている．

本実験では，タイプ 2b とタイプ 3 のコードクローンを抽出し．それぞれ不具合が含まれているかどうかを調査する．まず，対象ソフトウェアのコードクローンを検出する．コードクローンの検出は，ソースコードの正規化を行う前と行った後の 2 つのソースコードに対して行う．検出されたコードクローンに対して，提案手法を適用して差分を取得する．差分を取得するとき，ソースコードの正規化を行わずに検出したコードクローンに対しては，コードクローンの一致条件 1 を用いない．ソースコードの正規化を行って検出したコードクローンに対しては，コードクローンの一致条件 1 を用いる．したがって，タイプ 2b とタイプ 3 それぞれに対して，ソースコードの正規化，コードクローンの一致条件 1 を用いている場合と用いていない場合，計 4 種類の実験を行っている．また，good 値は 0.7 としている．

4.1.1　タイプ 2b のクローン抽出に用いたツール

タイプ 2b の抽出には，コードクローン検出ツールとして CCFinderX を用いた [6]．CCFinderX には P-Match を用いるか用いないかを選択するオプションが存在するため，P-Match を用いない場合から用いる場合の差分を取ることで，タイプ 2b のコードクローンを抽出できる．CCFinderX では，50 トークン以上のコードクローンを検出するように設定した．

4.1.2　タイプ 3 のクローン抽出に用いたツール

タイプ 3 の抽出には，コードクローン検出ツールとして Dude [5] と CCFinder [4] を用いた．全てのタイプを検出する Dude から，タイプ 2b までを検出する CCFinder の結果の差分を取ることで，タイプ 3 のコードクローンを抽出できる．Dude では 7 行以上のコードクローンを，CCfinder では 50 トークン以上のコードクローンを検出するように設定した．

4.2　結　　果

表 2 は，差分取得前と差分取得後それぞれのクローン行数とコード片の数を示している．クローン行数は，ソースコード上で重なっていても異なるコードクローンであれば重複して数えている．また，ソースコードの正規化を行うとソースコードの行数が減少するが，この表のクローン行数は，正規化前のソースコードに当てはめて行数を数えている．また，差分取得前の情報に関してはソースコードの正規化のみが影響しており，その結果検出されるコードクローンにも差が生じている．

いずれの場合でも，差分取得により大幅にクローン行数やコード片の数が削減できていることがわかる．また，ソースコードの正規化とコードクローンの一致条件 1 を用いることで，用いない場合に比べてより良い結果が出ていることもわかる．

表 2 における差分取得後のコードクローンの中に，実際に不具合があるかどうかを調査した結果が**表 3** である．**表 3** では，既に修正されている不具合の箇所，修正されていないが不具合の可能性が考えられる箇所，不具合ではないが可読性などの観点から修正すべき箇所を示している．

タイプ 2b のコードクローンは，ソースコードの正規化とコードクローンの一致条件 1 を用いることでより多くの不具合を検出できている．一方，タイプ 3 のコードクローンは検出された不具合の数が減少してしまっている．この理由については次章で考察する．

表 2　差分取得の効果

		クローン行数	コード片の数
正規化，一致条件 1 なし			
タイプ 2b	差分取得前	217,527	9,739
	差分取得後	18,374	945
タイプ 3	差分取得前	260,207	11,391
	差分取得後	19,574	647
正規化，一致条件 1 あり			
タイプ 2b	差分取得前	204,087	8,749
	差分取得後	14,355	661
タイプ 3	差分取得前	256,813	11,080
	差分取得後	16,773	556

表 3　検出された不具合の数

	修正済み不具合	不具合の可能性有り	不具合ではないが修正すべき
正規化，一致条件 1 なし			
タイプ 2b	4	5	5
タイプ 3	5	2	2
正規化，一致条件 1 あり			
タイプ 2b	6	5	9
タイプ 3	3	2	2

5. 考　　察

5.1 差分取得の効果

表 2 で示したように，差分を取得することでコードクローン量を大幅に削減することが確認できた．差分取得後のコードクローンを実際に不具合があるかどうか調査したが，いずれの場合もおよそ 10 時間程度の調査時間であった．差分取得前に調査を行っていた場合，単純計算でも 100〜150 時間程度の調査時間が必要になる．1 日 5 時間の調査を行った場合，2,30 日かかる調査が 2 日程度で済むことになるので，提案手法が優れているということがわかるであろう．

また，既存のコードクローン検出ツールの出力結果のみを使用した場合は，あるコードクローンにどのような不具合の可能性が高いかが分からなかった．しかし，タイプ 2b のコードクローンであれば，コード片間で対応の取れていない識別子に注目すれば良いし，タイプ 3 のコードクローンであれば，コード片間で異なっている文に注目すれば良い．提案手法を用いた場合はコードクローンのどこに注目すればよいか分かっているので，調査に要する時間的コストの削減が見込めると考えられる．

一方で，差分取得により削除されたコードクローンが多数存在している．削除されたコードクローンに多くの不具合が含まれていると，提案手法の有効性が薄れてしまうことになるため，削除されたコードクローン部分の調査が必要である．ただし，削除されたコードクローンの多くは，タイプ 1 など不具合を引き起こしにくいコードクローンであるため，この中に多くの不具合が含まれている可能性は高くないと考えている．

5.2 正規化，一致条件 1 の効果

5.2.1 タ イ プ 2b

表 3 を見ると，タイプ 2b のコードクローンに関しては，ソースコードの正規化やコードクローンの一致条件 1 を用いた方がより多くの不具合を検出できていた．

新たに検出できた不具合のうち 1 つは，正規化や一致条件 1 を用いない場合でも，コードクローン検出ツールの出力結果として検出されていた．しかし，差分を取得するときに，一致条件 1 の内包条件を満たしていないコードクローンが同一であると判断されてしまったために削除されていた．この事実により，コードクローンの一致条件 1 の有用性がわかるであろう．

もう 1 つは，正規化を用いない場合において，そもそもコードクローンとして検出されていなかった．CCFinderX はトークンベースでコードクローンを検出しているため，本来はソースコードの正規化によって検出されるコードクローンに影響は無いはずである．CCFinderX のオプションが何か影響を与えている可能性が考えられるが，今のところ原因は不明である．

5.2.2 タ イ プ 3

表 3 を見ると，タイプ 3 のコードクローンは，ソースコードの正規化やコードクローンの一致条件 1 を用いた方が検出される不具合の数が減少してしまっていた．検出されなかった 2 つのコードクローンを調査したところ，そもそも差分取得前のコードクローンとして検出されていなかった．これは，ソースコードの正規化に起因するものであった．

図 3 を見るとわかるように，ソースコードの正規化を行うとソースコードの行数が減少する．タイプ 3 の実験に用いた Dude は行をベースにコードクローンを検出するツールであり，ソースコードの行数が減少すると，元のソースコードでは検出できていたコードクローンが検出できなくなってしまうことがある．その結果として，正規化を行わない場合では検出できていたコードクローンが検出されていなかったのである．

したがって，行をベースにコードクローンを検出するツールを用いる場合は注意が必要である．正規化を行う場合は，正規化を行わない場合よりもコードクローンの最小一致行数を少なくするのが 1 つの解決策として考えられる．ただし，その場合はより多くのコードクローンが検出されることになるので，不具合検出の要求に応じて適切な値を設定する必要がある．

6. 関 連 研 究

Li らのコードクローン検出ツール CPMiner は，タイプ 3 のコードクローンも検出でき，バグ検出機能も備えている [7]．CPMiner は C，C++で記述されたプログラムに対して適用可能である．一方，我々の提案手法は，使用するコードクローン検出ツールが対応しているどのような言語にも適用可能である．

Li らは Linux2.6.6 に対して適用実験を行っているので，その結果と我々が提案した手法との違いを考察する．CPMiner は Linux2.6.6 全体に対して 49 個の不具合を検出している．本稿で行った適用実験との規模の差を考えると，検出可能な不具合の数は提案手法の方が CPMiner と同等もしくはそれ以上であり，不具合の検出数に関しては我々の提案手法の方が優秀である．しかし，我々の手法は不具合でない誤検出も非常に多いため，検出精度は CPMiner の方が優秀である．

その理由として考えられるのが，CPMiner が行っている *UnchangedRatio* を用いたフィルタリングである．*UnchangedRatio* は，コピーアンドペーストにおける 2 つのコード片において，ある識別子がどの程度変更されているかの割合を表す．コピー元のコード片に存在するある識別子の総数を *NumTotal*，その識別子がコピー先のコード片において変更されていない数を *NumUnchanged* とすると，*UnchangedRatio* は次の式で表される．

$$UnchangedRatio = \frac{NumUnchanged}{NumTotal}$$

UnchangedRatio の値は 0 から 1 で，$UnchangedRatio = 0$ のときはその識別子が全て変更されていることを示す．$UnchangedRatio = 1$ のときは，その識別子がまったく変更されていないことを示す．0 ではないが，0 に近い値であると，コード片全体に対して変更されていない識別子が少しだけ存在することになるため，修正漏れの可能性が非常に高くなる．

CPMiner は，このフィルタリングを行うことでより不具合の可能性が高いコードクローンを検出するようにしている．一方，我々の提案手法はこのようなフィルタリングを一切行っていないので，より誤検出が多くなってしまっていると考えられる．

Jiang らは，自身らが作成したコードクローン検出ツール Deckard を用いて不具合を検出する手法を提案している [8]．Jiang らは，コードクローンになっている箇所を囲むような条件文に注目して不具合の検出を行っている．それら条件文が異なるものであったり，条件文の述部に不整合があるコードクローンを検出する．また，コードクローン間で識別子の修正漏れなどがある場合も不具合として検出している．

Jiang らの手法でも，誤検出を減らすためにフィルタリングを行っている．コードクローンを囲んでいる条件文が異なっている場合は不具合として検出するが，for 文と while 文，switch-case 文と if-else 文など，意味的に同じような処理になるものは除去している．また，条件文の述部についても，$e1 < e2$ と $e2 > e1$ のように意味的に同じものは除去している．さらに，識別子の修正漏れがある場合でも，修正漏れの数が多い場合はあえて修正していない可能性が高いため除去している．

Jiang らの手法は，Linux2.6.19 全体に対して 41 個の不具合を検出している．Linux のバージョンが異なるので正確な比較は行えないが，我々が実験を行った Linux2.6.6 よりも規模が大きいので，不具合の検出数に関しては我々の提案手法の方が優秀である．しかし，Jiang らの手法もフィルタリングを行っており，検出精度は Jiang らの手法の方が優秀である．また，我々の提案手法や CPMiner との違いとして，不具合だけではなくプログラミングスタイルによる問題点も検出可能である．

7. まとめ

本稿では，コードクローン検出ツールの出力結果の差分を取ることで，不具合の可能性が高いタイプのコードクローンのみを取り出し，不具合の検出を容易にする手法を提案した．提案手法の適用実験を行ったところ，非常に短時間で不具合を検出できることが確認された．

今後の課題としては，まず差分取得によって削除されたコードクローンの調査が挙げられる．削除されたコードクローンにはどの程度の不具合が含まれているか，含まれているとすればそれはどのような原因による不具合か，などといったことを調査することで，提案手法の有用性を確かめることができる．

本稿では，行の情報しか持っていないコードクローン検出ツールにも対応するため，コードクローンを開始行，終了行を用いて表現していたが，それをトークン情報に変換することができれば差分取得の精度をより高めることが可能になると考えられる．行数を用いている場合，1 行の増減によって good 値の値が大きく変わってしまう可能性があるが，トークン情報を用いることでより正確にコードクローンの一致判定ができるようになるので，より精度を高めることができるであろう．

さらに，差分取得後のコードクローンに対してフィルタリングを適用することで，より精度を高めることもできるかもしれない．適用実験において検出されたコードクローンには，同じようなコードの繰り返しを多く含むものや，デバッグ用コードの有無によってタイプ 3 のコードクローンと検出されているものが見受けられた．このようなコードクローンは不具合の可能性が低いため，これらを削除するようなフィルタリングを作成することで，さらに精度を高めることも考えている．

謝辞

IJARC(マイクロソフト産学連携機構) 第 3 回 CORE プロジェクト，文部科学省 科学研究費補助金 基盤研究 (A)(課題番号:17200001)，および若手研究 (スタートアップ)(課題番号:19800022) の助成を得た．

文献

[1] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, 33(9):804–818, Sep 2007.

[2] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. *Proc. of the International Conference on Software Maintenance*, pages 109–118, Aug 1999.

[3] B.S. Baker. Finding clones with dup : Analysis of an enperiment. *IEEE Transactions on Software Engineering*, 33(9):608–621, Sep 2007.

[4] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder : A multi-linguistic token-based code clone detection system for largescale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, Jul 2002.

[5] R. Wettel and R. Marinescu. Archeology of code duplication : Recovering duplication chanins from small duplication fragments. *Proc. of the 7th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 63–70, Sep 2005.

[6] CCFinderX. http://www.ccfinder.net/ccfinderx-j.html.

[7] Z. Li, S. Myagmar S. Lu, and Y. Zhou. Cp-miner : Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32(3):176–192, Mar 2006.

[8] L. Jiang, Z. Su, and E. Chiu. Context-based detection of clone-related bugs. *Proc. 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE '07)*, pages 55–64, Sep 2007.