実務で使われるコードクローン検出・追跡システムをめざして

三木 聡　大歳 始　浅原 明広　大澤 俊晴　千葉 滋

本発表では我々が実務で使われる商用システムとして開発しているコードクローンの検出および追跡をおこなうシステムについて述べる．コードクローンはプログラム中に現れる重複コードのことである．これはコード断片のコピー＆ペーストによって生まれ，ソフトウェアの品質低下の要因となることが知られている．コードクローン検出器はいくつも発表され，無償で利用できるものもあるが，産業界の開発現場に広く普及しているとは言いがたい．我々は，特別な設定なくインストールしてすぐに使え，日々の開発のバックグランドで動かすだけで，コードクローン検出の効果を実感できるようなツールを目指して，コードクローンの検出・追跡器を開発している．本発表では実務で使ってもらうための工夫や，発見したコードクローンをその後も追跡し続けて変更の量からその危険度を判断する機能などについて述べる．

1　はじめに

コードクローン，あるいは重複コード，は，プログラム中のコードの断片を別のところにコピー・ペーストすることにより作り出される．長年の研究により，コードクローンの存在はプログラムの品質に大きな影響を与えることが知られており，それを検出するツールも数多く開発されてきた．

しかしながら実務でのソフトウェア開発の現場を見ると，コードクローンを検出してリファクタリングで除去し，プログラムの品質を維持する，という活動が広く実践されているとは言いがたい．コードクローンや重複コードという概念自体も広く知られているわけではない．それにはいくつかの理由があるものの大きな理由の一つは，実務での開発ではコードクローンを見つけても，リファクタリングでそれを除去することが現実的にしばしば困難だから，と我々は考えている．見つけても何もできなければ，コードクローンを見つけようという気持ちも生まれない．

本稿では我々が商用製品として開発しているコードクローン検出器 CloneTracker について述べる．この検出器はコードクローンの「修正もれ」の発見に重点を置いている．修正もれとは，コードクローンをなすコード片の一方にバグ修正等を加えたとき，他方のコード片に同様の修正を加え忘れることである．CloneTracker は，検出したコードクローンが過去から現在までにどのように修正されてきたかを追跡し，修正もれの危険が高いコードクローンを優先して提示する．これによって，検出したコードクローンをリファクタリングの労を払って除去しなくても，コードクローン検出の意義を感じられるツールにすることを狙っている．プログラムの開発履歴からコードクローンの修正もれを検出するツールは過去にも開発されてきたが，CloneTracker は継続的にコードクローンを追跡し続けるのが特徴である．また経験から得た様々な重み付けの規則を用いて，実務で開発する技術者やプロジェクトマネージャー (PM) にとって有用であると思われるコードクローンを検出し，提示するように工夫している．この重み付けの規則には対象のプログラミング言語に特化した規則も含まれる．

なお CloneTracker は現在も開発中である．本稿で

---

* Toward a code clone detection/tracking system used in industry. (This is an unrefereed paper. Copyrights belong to the Authors.)

Satoshi Miki, 株式会社フィックスターズ, Fixstars Corp..
Hajime Otoshi, Akihiro Asahara, Toshiharu Osawa, Shigeru Chiba, 株式会社 Sider, Sider Corp..

述べる CloneTracker は本稿執筆時点でのものであり，今後の開発にともない変わる可能性がある．

2 産業界におけるコードクローン検出

コードクローンは，その重複具合によっていくつかの種類に分類できる [1]．古典的な分類は Type 1 から 4 までの 4 種類である．Type 1 は完全に一致するコード片，Type 2 は変数名等が異なる他は一致するコード片，Type 3 はほぼ同一のコード片，そして Type 4 は意味的によく似ているが制御構造やアルゴリズムが異なるコード片，の組である．コードクローンはプログラムの品質を考える上で重要なものとされ，それを検出するツールが数多く開発されてきた．例えば CCFinder [5]，NiCad [8]，SourcererCC [9] などの検出器がある．

どこまで異なるコード片の組を Type 3 コードクローンと呼ぶかについて明確な合意はないと思われるが，例えば CCAligner [10] は large-gap クローンと呼ぶ種類の Type 3 コードクローンの検出器である．Large-gap クローンとは，一方のコード片の途中に多数の文（あるいは行）を挿入・削除したものがもう一方のコード片だと見なせるようなコード片の組である．多数の文を挿入・削除するのがコード片の途中の一ヶ所だけでなく，複数箇所である場合を large-variance クローンと呼び，それを検出するできるようにした検出器 LVMapper [11] や NIL [7] も開発されている．

2.1 コードクローン検出の目的

実務上，Type 1 および Type 2 コードクローンの検出は，誤って製品に混入した自社が権利を持たないプログラムの発見に役立つ．例えば 2000 年代後半には，オープンソース・ライセンスのプログラムの一部を，知らぬ間に無知な技術者が自社製品のプログラムに流用してしまい，結果としてその製品がライセンス違反となる事件が頻発した．これを未然に防ぐため，オープンソース・ライセンスのプログラムとの間のコードクローンを製品のプログラムから検出する Protex [13] などの製品が販売されている．

Type 3 コードクローンは実務におけるソフトウェア開発で最も頻繁に現れるコードクローンであると考えられ，その検出はプログラムの品質向上に役立つ．すでにある関数と類似した機能をもつ関数を実装するときに，元の関数の宣言をコピーしてプログラム中の別の場所に貼り付け，目的に合うように少し修正する，というのは一般的によくおこなわれる作業であるが，これが Type 3 コードクローンを生む．このようなコードクローンは，生まれた直後は比較的無害と考えられるが，バグや機能拡張などの理由でコードの修正が必要になった際に，片方のコード片だけを修正して，もう片方のコード片の修正を忘れる危険性をはらむ．実際に片方の修正を忘れる「修正もれ」がおきると，大きな障害を引き起こしかねない．

```
1 if (ERR_OK == ret) {
2   /* send message */
3   ret = queue_snd(&msg,
4                   sizeof(struct _msg));
5 }
```

リスト 1　元のコード片

```
1 if (ERR_OK == ret) {
2   /* receive message */
3   ret = queue_rcv(&msg,
4                   sizeof(struct _msg));
5 }
```

リスト 2　コピー&ペースト後

```
1 if (ERR_OK == ret) {
2   /* receive message */
3   ret = queue_rcv(&msg,
4                   sizeof(struct _msg));
5   ret = new_return_code(ret)
6 }
```

リスト 3　修正もれ

例えばリスト 1 がコピーされて修正され，リスト 2 が書かれたとする．呼ばれている関数は異なるが，この二つのコード片は Type 3 コードクローンである．その後，何らかの理由でライブラリの仕様が変わり，リスト 2 に修正が加えられてリスト 3 になったとする．5 行目の関数呼び出しが修正で追加された行である．このときリスト 1 にも同様の修正が加えられていなければ，おそらく修正を忘れたバグと考えられ

る．これが「修正もれ」である．

全てのコードクローンが将来の修正もれをおこすわけではないが[6]，危険を避けるためにコードクローンは見つかり次第リファクタリング作業で取り除くことが望ましい．リファクタリングをせずに済むように，見つけたコードクローンを追跡し，そのコードクローンに修正もれを検知すると，修正からもれたコード片も同様に修正するように警告を発するツールも開発されている[3][12][4]．

2.2 普及しないコードクローン検出

このような研究の蓄積にもかかわらず，コードクローンを積極的に発見してソフトウェアの品質改善に役立てるという姿勢が，実務としてのソフトウェア開発の現場に十分浸透しているとはいいがたい．コードクローン検出器の利用も普及しているとは言えない．我々の観察によれば，とくに日本国内で業務としてソフトウェア開発をおこなっている技術者（エンジニア）の間の，コードクローンあるいは重複コード (duplicated code) の認知度は低い．

コードクローン検出が実務の開発で普及していないのは，実際のところ，コードクローンが検出されても技術者はその扱いに困るからだと，見込み顧客との対話から我々は考えている．一般に，検出されたコードクローンは除去するべきである．しかし通常のリファクタリングでは簡単に除去できないコードクローンが多いことも知られている[6]．多大な労力を払ってコードクローンを除去するべきか，それが将来問題となる危険性は低いと予想して放置すべきか，開発全体の人的・時間的制約まで含めて考えると悩ましい．開発しているのが金融や官公庁向けのシステムの場合，万一リグレッションが起きたときの社会的影響が大きく，後からのリファクタリングは気軽に実施しにくい．結局，検出されたコードクローンをそのまま放置せざるを得ないことが多いが，そうであるなら何のためにコードクローンを検出するのか，ということになる．

上記の問題に加え，オープンソース・ソフトウェアとして自由に利用可能なコードクローン検出器については，利用の際の敷居の高さも指摘されている[14]．まず，実務での開発によく用いられる Windows 環境で検出器を動かすのが容易ではない．また，検出結果を見やすくするには他の可視化ツールと組み合わせる必要があり，手軽ではない．さらに，チームが開発しているプログラムの品質向上に本当につながるコードクローンを検出するには，検出器の検出パラメータを慎重に調整する必要がある．しかし，コードクローン検出の研究者や専門家でない，一般の技術者にとって，このような検出パラメータの調整は困難である．これらの問題点については我々も同様の意見を見込み顧客から得ている．

単独の商用製品として提供されているコードクローン検出器で，よく知られたものは我々の知る限り存在しない．存在するのは，プログラムの静的解析の一部としてコードクローン検出をおこなう製品である．例えば SonaSource 社の SonarQube[†1] はプログラムの品質改善のために様々な解析をおこなうツールだが，コードクローンの検出機能を備える．同様の静的解析ツールに Synopsys 社の Coverity[†2] がある．これらの製品に含まれるコードクローンの検出機能がどのくらい使われているかは不明だが，これらの製品があるので他のコードクローン検出器は不要であるという声も我々は知らない．

コードクローン検出器が実務での開発で活用されない，もう一つの理由はプロジェクトマネージャー (PM) の無関心にもあると我々は考えている．コードクローン検出器のようなツールを開発チームに導入する際の決定権をもつのは PM である．開発しているプログラムの品質に責任を負うのは PM であるが，とくに大規模な開発チームの PM の中にはプログラムの品質にあまり関心がなかったり，関心があってもプログラミングの技術的詳細に疎く，ツールの導入は実際の開発をおこなう協力会社が考えればよい，と考える者がいる．そのような PM の開発チームの場合，コードクローン検出器を導入して積極的に活用しようという気運は生まれにくい．

このように現状でコードクローン検出は実務として

†1 https://www.sonarsource.com
†2 https://www.synopsys.com

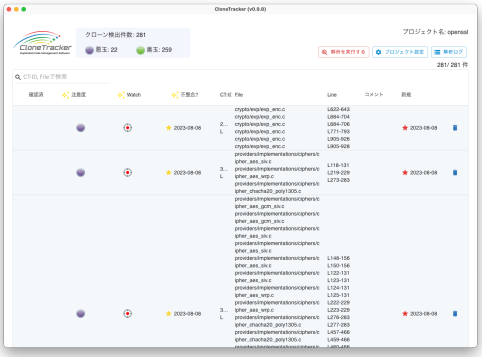

図 1　CloneTracker による解析結果

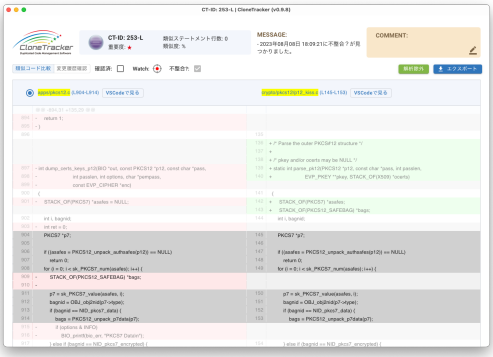

図 2　検出されたコードクローンの表示

のソフトウェア開発の現場で広く行われているとは言いがたい．しかし，コードクローンの存在に注意を払うことがプログラムの品質向上につながることは様々な研究から明らかであり，コードクローン検出に一般の技術者が無関心である現状はけして望ましくない．このような状況を打開するためには，検出器を使った PM や技術者に「これは有用だ」と思う体験（いわゆる aha/wow moment [2]）を届けるクローン検出器を開発し，その意識を変えていくことが必要である．

3　CloneTracker

我々は前章で述べた問題意識の下，コードクローン検出器 CloneTracker†3 を開発している．これは商用製品として単体で提供されるクローン検出器で，`https://clonetracker.com` から入手可能である．

3.1　過去から未来へ継続的な追跡

CloneTracker の使い方は簡単である．インストール後，解析対象のプログラムが存在するディレクトリを指定するだけである．それを指定すると解析が始まり，終了すると検出されたコードクローンの一覧が表示される（図 1）．詳細を調べたいコードクローンを選択すると，コードクローンと認識されたコード片が左右に並べて表示され，差分が強調表示される（図 2）．

CloneTracker は検出したコードクローンの変更履歴を過去から未来にかけて継続的に追跡し，「修正もれ」の危険性が高いコードクローンを中心に，重要なもの，PM や技術者が見たいはずのコードクローンを優先して提示するツールである．このツールは検出したコードクローンをリファクタリングで除去するように利用者に無理に求めない．除去が難しいコードクローンはそのままにしてよい．そのままにしても，コードクローンの存在をツールが記録・追跡するので，将来，修正もれが起きないように利用者は気を配ることができる．仮に気づかず「修正もれ」をおこしても，ツールがそれを知らせる．CloneTracker は，本来は悪いものであるコードクローンと無理な労力をかけずに上手に付き合っていくためのツールとも言える．またこれにより，CloneTracker はリファクタリングの労力を払わずとも，動かすだけで利用者が有用性を感じられるツールとなることを狙っている．

3.2　善玉クローンと悪玉クローン

継続的なコードクローンの追跡のため，CloneTracker が解析するプログラムは git†4 で管理しなければならない．CloneTracker が初めて対象のプログラムを解析するときは，その最新版だけでなく，過去の版（90 日前の版と，90 日前の版と最新版の中間の版）も合わせて解析をおこない，検出されたコードクローンの変化の履歴を調べる．

CloneTracker は新しいコミット (commit) がある

†3　文献 [3] の CloneTracker とは異なる．

†4　`https://git-scm.com`

とコードクローン検出をおこない，前回の検出結果との差分からコードクローンの変化を調べる．新しいコミットが，既存のコードクローンの一方または両方のコード片の，直前またはその内部に新しい行を挿入している場合，または行を削除している場合，それは「修正もれ」の可能性が高いと判断する．つまりバグや機能拡張の理由でコードに修正が必要なのに，そのコードの全てのコピーに修正が加えられていないと判断する．その判断が正しければ，明らかなバグであるので，CloneTracker はそのコードクローンを「悪玉クローン」に分類して注意を促す．

一度，悪玉クローンに分類されたコードクローンは，その後もずっと悪玉である．次のコミットで抜けていた修正が追加されて「修正もれ」が修正されても，「善玉クローン」にはならず悪玉のままである．これは今後も，一方のコード片が修正されたら合わせて他方も修正しなければならない，注意を払うべきコードクローンと予想されるからである．むしろ「超悪玉クローン」と分類されるべきものである．悪玉クローンが悪玉でなくなるのは，リファクタリングによってそのコードクローンが除去されて消滅したときである．

コードクローンの中には，やがて各コード片に異なる修正が加えられて，最後には互いに似なくなり，コードクローンではなくなるものが多いことが知られている [6]．しかし CloneTracker は，悪玉クローンに分類したコードクローンがその後に多くの修正が加えられて相当異なるクローンになってしまっても，悪玉クローンとして追跡を続ける．これはコードクローンが生まれた後，それぞれのコード片に Goolge Analytics[†5] によるアクセス解析や Sentry[†6] による障害監視のサービスを利用するためのコードが挿入されると，見かけ上大きく異なってしまい，コードクローンでなくなることがあるからである．このような非機能要件に関連するコードを除外してコード片を比較するとコードクローンと判断されるコード片は少なくない．

†5 https://analytics.google.com
†6 https://sentry.io

3.3 コードクローン検出

CloneTracker のコードクローン検出は CCFinder [5] や CCAligner [10] と同様，プログラムのトークン列に対しておこなっている．トークン化のための字句解析器には各言語の標準的なコンパイラのものを移植して使っているので，何をトークンと見なすかはその字句解析器に依存する．例えば Java の場合，`public` や `class` のような予約語はそれぞれ異なるトークンであるが，変数名やメソッド名などは識別子としてすべて同一のトークンとなる．なお C/C++ の場合，字句解析はマクロ展開などの前処理をする前のプログラムに対しておこなう．その際，`#define` のような前処理指令は無視される．

次に，得られたトークン列をスライド窓 (sliding window) 方式で 5 行ずつ取り出しハッシュ値を計算する（5 は我々が経験的に得た値である）．このとき `{` や `}` のような括弧やカンマ `,` やセミコロン `;` だけの行は無視し，5 行に含めない．そしてプログラム全体でこのハッシュ値が一致する 5 行の組を探す．ハッシュ値が一致する 5 行の組のうち，さらにその中央行の生テキスト（字句解析前の文字列，ただしコメントと空白文字は除く）が一致するとき，その 5 行の組をクローンと認識する．スライド窓とハッシュ値を用いてコードクローンを探す方式は CCAligner と似ているが，CloneTracker は 5 行のハッシュ値が完全に一致しないとコードクローンと認識しない．一致度の低いコードクローンを検出してもあまり有用でないと我々は考えるからである．

5 行のハッシュ値を計算するときに括弧等だけの行を除外するのも，あまり有用でないコードクローンの検出を抑制するためである．例えばトークン列に対して検出をおこなうと

```
    1, 2, 3,
  ]});
```

のような配列や構造体の初期化部分をコードクローンとして検出してしまいがちだが，これは検出してもあまり有用ではないことが多い．あるいは誤検出と判定されるべきものである．

5 行の組のクローンと認識した後，5 行以上の長さ

のコードクローンを検出するため，互いにクローンと認識された 5 行の組のうち，連続して現れるものを連結する．例えば 5 行の組 A と A’ と，別の 5 行の組 B と B’ とが，それぞれクローンと認識されたとする．A と B，そして A’ と B’ がそれぞれ同じソースファイル中の連続する行に現れる場合，A と B，A’ と B’ を連結し，AB と A’B’ の組がクローンであると認識する．このとき A と B，A’ と B’ の間は必ずしも完全に連続する必要はなく，A と B，A’ と B’ の間に一致しない行が 3 行まで含まれていても許容する．

このようにして見つけたコードクローンは，重要度が高そうなものだけを提示するため，重み付けされる．CloneTracker の利用者に提示されるのは一定の閾値以上の重みをもつコードクローンだけである．重み付けの規則は我々が経験的に得たもので例えば次のようなものである．

- より行数の多いコードクローンほど重要．
- 生テキスト同士で比較したときに一致部分が多いほど重要．
- `if` や `while` など制御構造に関わるトークンが多いほど重要．
- 同じトークン列の行が繰り返されるコードクローンは重要でない．
- 特定のトークン列が頻繁に現れるコードクローンは重要でない．例えば C 言語の，識別子，識別子，`=`，数値，`;` というトークン列は `int a = 1;` のような変数宣言である．これを多く含むコードクローンは重要でないことが多い．

3.4 適用事例

2.1 節のリスト 1，2，3 の例は，実際に CloneTracker が我々の顧客のプログラムから発見した事例である．ただし守秘義務により実際のコード片を簡略化している．この事例では，この後のコミットで実際にリスト 1 も 3 と同様に修正されている．CloneTracker は，このコードクローンは今後も修正が加えられるときは両方のコード片に同時に加えなければならないと認識し，「超悪玉クローン」に分類している．

リスト 3 ではコード片の内部に新しい行が挿入さ

```
1 ret = new_return_code(ret)
2 if (ERR_OK == ret) {
3   /* receive message */
4   ret = queue_rcv(&msg,
5                   sizeof(struct _msg));
6   ret = new_return_code(ret)
7 }
```

リスト 4　修正後のコードクローン

れているが，片方のコード片の直前に新しい行が挿入されたときも CloneTracker は修正もれの可能性ありと判断する．しかしコード片の直前に挿入された行が，そのコード片の修正の一部か否かを判定するのは実は容易ではない．例えばリスト 3 の事例で，実際に CloneTracker が修正後のコードクローンとして検出したのはリスト 4 であった．if 文の直前に関数呼び出しの行（1 行目）が挿入されたと判定している．該当するプログラムの部分を調べたところ，1 行目はこのコード片の前の部分に属すと判断でき，1 行目をこのコード片に対する修正とした CloneTracker の判定は厳密には誤りであった．現在，CloneTracker は直前に挿入された文が変数宣言であるならコード片の修正の一部と見なさないが，if 文であるときは見なす，といった我々の経験から得た規則で判定をしている．

このように修正もれの判断は容易ではない．対象となるプログラミング言語の意味論や慣用句を考慮して判断しなければならないこともある．例えば Java のプログラムには以下のような慣用句と呼べるコード片が頻繁に現れる．

```
public static void main(String args[]) {
  Options opts = Options.parseArgs(args);
```

これは Apache Cassandra[†7] の一部である．このようなコード片をコードクローンだと検出しても有用とは言えない．慣用句なので重みを下げなければならない．さらにこれをコードクローンと認識した場合，片方のコード片だけが

```
public static void main(String[] args) {
  Options opts = Options.parseArgs(args);
```

と修正され，引数 args の型の書き方が変わったとす

†7 https://cassandra.apache.org

る．このとき，他方のコードの型の書き方が変わっていないことを，修正もれと判断するか否かは難しい．修正の前後でプログラムの意味は変わらないので，修正がもれていても大きな問題ではないからである．

このように CloneTracker を実務で有用なコードクローン検出器とするためには，対象プログラミング言語ごとに細かな調整が必要である．このため現在の CloneTracker は，実装上は他の言語のプログラムも分析はできるものの，公式には C/C++ および C# だけに対応している．

3.5 実装

開発の過程では紆余曲折があったが，現在の CloneTracker は各技術者の開発用 PC にインストールして使うスタンドアロンのソフトウェアである．クラウド上の SaaS (Software As A Service) として実装されていた時期もあったが，ネットワーク接続を必要としない製品となっている．実務の開発では開発中のプログラムを外部の SaaS に転送することがセキュリティの観点から禁止されていることが多いからである．また対応するオペレーティングシステムは現在 Windows と macOS および Linux である．

CloneTracker の実装に利用しているフレームワークは Tauri[†8] である．ユーザインターフェースとコードクローン検出の主要部は TypeScript で実装されており，残りの部分は Rust で実装されている．当初用いていた Electron[†9] から Tauri に変更することで，ファイルシステムやデータベース（SQLite[†10] を利用）へのアクセスを Rust で行えるようになり実装の自由度が増した．一方で Tauri は各 OS 標準の web レンダラを利用するので，その差異に注意を払う必要がある．コードクローン検出のための字句解析器は Clang や Javac の字句解析器を Emscripten[†11] で WebAssembly にコンパイルして実行している．これは当初の実装で Electron を利用していたためである．コードクローン検出の主要部が TypeScript で実装されているのも同じ理由で，Rust で再実装することを計画している．

表 1　初回の解析時間（三つの版の解析時間の合計）

	Apache	1MLOC	OpenSSL
Core i9	2m 55s	3m 26s	9m 55s
Core i7	4m 33s	6m 29s	17m 08s
Apple M2	5m 54s	9m 29s	28m 53s
行数	351,414	1,445,144	736,033
ファイル数	583	12,381	1,984

CloneTracker がプログラムの解析に要する時間を表 1 に示す．Apache (Apache HTTP Server[†12]) と OpenSSL[†13] は C 言語のプログラムである．1MLOC は文献 [7] で用いられているデータセット [†14] で，Java のプログラムである．Core i9 は i9-12900 (2.4GHz) および 32GB メモリの，Core i7 は i7-1165G7 (2.8GHz) および 16GB メモリの Windows マシンである．Apple M2 は 8GB メモリの MacBook Air (macOS) である．

表の解析時間は初回のそれなので，最新版のプログラムだけでなく，90 日前の版，および，最新版と 90 日前の版の中間の版の計三つのプログラムを解析した合計の時間である．ただし 1MLOC は最新版のみであるので一回分だけの解析時間である．コードクローンを継続的に追跡するためのデータベースの作成なども含むこと，現在の実装が単一スレッドであること，などからどの場合も比較的時間がかかっている．

4 関連研究

我々は CloneTracker を独自に開発しているが，コードクローンの検出手法に大きな独創性があるわけではない．CloneTracker の特徴は，実務で開発している技術者や PM にとって有益なコードクローンを提示するように，個々の具体例に対応した工夫をこらしている点である．とくにコードクローンの変更の履歴を追跡し，修正もれによる不具合を防止することに注力している．

コミットの履歴を参照してコードクローンの「修

†8 https://tauri.app
†9 https://www.electronjs.org
†10 https://www.sqlite.org/index.html
†11 https://emscripten.org
†12 https://github.com/apache/httpd
†13 https://github.com/openssl/openssl
†14 https://doi.org/10.5281/zenodo.4491208

正もれ」等を指摘する手法は CCEvovis [4] でも提案されている．しかし論文からは，CCEvovis は二つのコミット間の違いの分析に留まっていると読めるが，CloneTracker は三つ以上のコミットにわたって，より長期間コードクローンを追跡する．

文献[3] の CloneTracker もコードクローンの「修正もれ」を指摘するが，これは統合開発環境 Eclipse のプラグインである．git のような版管理システムとは統合されておらず，コードクローンの一方のコード片をエディタで編集しようとしたとき，他のコード片も合わせて修正するように促す機能をそなえる．我々の CloneTracker は逆にエディタとは統合されておらず，版管理システムと連携させて使うツールである．

5 まとめ

本稿では我々が商用製品として開発しているコードクローン検出器である CloneTracker について述べた．解析対象のプログラムが時間とともに修正，拡張される中で，コードクローンの変化を追跡するのが特徴である．CloneTracker は現在も実務において有用なツールとなるように開発中であり，他のコードクローン検出器との定量的な比較は今後の課題である．

参 考 文 献

[1] Bellon, S., Koschke, R., Antoniol, G., Krinke, J., and Merlo, E.: Comparison and Evaluation of Clone Detection Tools, *IEEE Transactions on Software Engineering*, Vol. 33, No. 9(2007), pp. 577–591.

[2] Bush, W.: *Product-Led Growth: How to Build a Product That Sells Itself*, Product-Led Institute, 2019.

[3] Duala-Ekoko, E. and Robillard, M. P.: Clonetracker: Tool Support for Code Clone Management, *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, New York, NY, USA, ACM, 2008, pp. 843–846.

[4] Honda, H., Tokui, S., Yokoi, K., Choi, E., Yoshida, N., and Inoue, K.: CCEvovis: A Clone Evolution Visualization System for Software Maintenance, *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, 2019, pp. 122–125.

[5] Kamiya, T., Kusumoto, S., and Inoue, K.: CCFinder: a multilinguistic token-based code clone detection system for large scale source code, *IEEE Transactions on Software Engineering*, Vol. 28, No. 7(2002), pp. 654–670.

[6] Kim, M., Sazawal, V., Notkin, D., and Murphy, G.: An Empirical Study of Code Clone Genealogies, *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, New York, NY, USA, ACM, 2005, pp. 187–196.

[7] Nakagawa, T., Higo, Y., and Kusumoto, S.: NIL: Large-Scale Detection of Large-Variance Clones, *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, New York, NY, USA, ACM, 2021, pp. 830–841.

[8] Roy, C. K. and Cordy, J. R.: NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization, *2008 16th IEEE International Conference on Program Comprehension*, 2008, pp. 172–181.

[9] Sajnani, H., Saini, V., Svajlenko, J., Roy, C. K., and Lopes, C. V.: SourcererCC: Scaling Code Clone Detection to Big-Code, *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, New York, NY, USA, ACM, 2016, pp. 1157–1168.

[10] Wang, P., Svajlenko, J., Wu, Y., Xu, Y., and Roy, C. K.: CCAligner: A Token Based Large-Gap Clone Detector, *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, 2018, pp. 1066–1077.

[11] Wu, M., Wang, P., Yin, K., Cheng, H., Xu, Y., and Roy, C. K.: LVMapper: A Large-Variance Clone Detector Using Sequencing Alignment Approach, *IEEE Access*, Vol. 8(2020), pp. 27986–27997.

[12] Yamanaka, Y., Choi, E., Yoshida, N., Inoue, K., and Sano, T.: Applying clone change notification system into an industrial development process, *2013 21st International Conference on Program Comprehension (ICPC)*, 2013, pp. 199–206.

[13] ブラック・ダック・ソフトウェア: オープンソースソフトウェア (OSS) ライセンス コンプライアンス ソリューション日本語版 Protex, `https://kyodonewsprwire.jp/release/201309264898`.

[14] 松島一樹, 小池耀, 井上克郎: CCX: SaaS 型コードクローン分析システム, コンピュータ ソフトウェア, Vol. 39, No. 4(2022), pp. 129–143.