特別研究報告

題目

ソフトウェア保守のためのコードクローン情報検索ツール

指導教官
井上 克郎 教授

報告者
泉田 聡介

平成 15 年 2 月 19 日

大阪大学 基礎工学部 情報科学科

平成 14 年度 特別研究報告

ソフトウェア保守のためのコードクローン情報検索ツール

泉田 聡介

内容梗概

コードクローンとは, プログラムテキスト中の同一, あるいは, 類似したコードの断片を意味する. コードクローンは,「コピーとペースト」によるプログラミングや意図的に同一処理を繰り返して書くということにより, プログラムテキスト中に作りこまれる. 一般に, コードクローンはソフトウェアの保守性を阻害する要因と言われている. 例えば, 機能追加を行う場合, 機能追加箇所に対するコードクローンを全て検出し, 必要があればその全てに対して同様の追加を行う必要がある. しかし, 大規模なプログラムの場合, コードクローンを全て検出することは困難である.

我々の研究グループでは, ソースコード中のコードクローンを検出するツール (CCFinder) の開発を行ってきている. CCFinder は, 大規模ソフトウェアの保守・デバッグ作業の支援, あるいは教育環境におけるプログラム評価支援を目的としている. しかし, コードクローンの検出結果はコードフラグメント対の位置情報であるため, 検出結果を直感的に理解することは困難であることが指摘されている.

本研究では, 保守作業を対象として, 効率的にコードクローン情報を抽出し, ユーザに提示することを支援するためのコードクローン情報検索ツールの試作を行う. 検索ツールに対して, コード片と検索対象ファイル群を与えることで, コード片に対するクローンを含んだファイルリストが表示される. 実際に, 検索ツールをあるソフトウェアプロジェクトにおける修正事例に対して適用した. その結果, 本ツールを用いることで, ツールを用いない場合よりも効率よく, 修正箇所の発見が行えることを確認した.

主な用語

コードクローン
ソフトウェア保守

目 次

1　まえがき **3**

2　コードクローン検出 **5**

2.1　コードクローンと既存のコードクローン検出法 5

2.2　コードクローン検出ツール CCFinder 7

2.2.1　概要 7

2.2.2　コードクローン検出処理手順 10

2.2.3　検出例 10

3　クローン情報検索ツール **12**

3.1　概要 12

3.2　概要設計 13

3.3　機能設計 14

3.3.1　検索フォーム 14

3.3.2　検索結果出力表示フォーム 15

3.4　実装 16

3.4.1　検索フォーム 17

3.4.2　検索結果の表示 17

4　適用 **21**

4.1　実行時性能 21

4.2　「かんな」への適用 21

4.2.1　修正に関するデータ 21

4.2.2　結果とその分析 23

5　まとめと今後の課題 **25**

謝辞 **27**

参考文献 **28**

1　まえがき

近年, ソフトウェアシステムの大規模化, 複雑化に伴い, プログラムの保守・デバッグ作業に要するコストが増加してきている. ソフトウェア保守を困難にしている一つの要因としてコードクローンが指摘されている.

コードクローンとは, ソースコード中に含まれる同一または類似したコード片のことである. それらは多くの場合, 既存システムに対する変更や拡張時における「コピーとペースト」による安易な機能的再利用の際に発生する. しかしながら, もしあるコード片にフォールトが含まれていた場合, そのコード片に対する全てのコードクローンについて修正を行わなければならない. また, 保守性を高くするため, 同値類を一つのサブルーチン等にまとめるのがよい場合もあるかもしれない. そのためには, コードクローンを全て検出することが必要となる.

そこでこれまで様々なコードクローン検出法が提案されている. 我々の研究グループもトークン単位でのコードクローンを検出するツール (CCFinder[15]) を開発してきており, これまで様々なソフトウェアに対する適用を行った. これらの適用の中で,CCFinder は大規模なソースコードであっても, 細粒度でのコードクローン解析を非常に高速に行うことが可能であると評価されてきた.

しかし,CCFinder から検出されてくるコードクローン情報は, ソフトウェア規模が大きくなればなるほど, 当然膨大なものとなる. ソースコード中から手で類似部分をもれなく探す手間がなくなったとはいえ, 現実的に, これらの膨大なコードクローン情報のそれぞれが, 何らかの有効な情報であるかどうかをひとつひとつ目視チェックしていくことは不可能である. そこで, 我々はさらに CCFinder の解析結果の参照支援システムを試作し [21][22][23], コードクローンの位置情報の視覚化と, コードクローンに対する単純的な評価尺度を用いてソースコードの参照支援を試みた. そのシステムにおいては, 規模の小さなソフトウェアであれば, 視覚化されたコードクローンの位置関係や, そのコード片の長さや数等で着目すべきコードクローンが容易に判別可能であった. しかし, 大規模ソフトウェアともなると, 個々の特徴は膨大な情報の中に埋没し, 実際の開発保守現場での利用は困難であった. こういった実用的な立場から,CCFinder とその参照支援システムには大規模ソフトウェアに対するコードクローン情報の識別性, 有意性が欠如していることが問題として取り上げられていた. 当然, 識別性, 有意性を確保していくためには, なんらかの利用目的の策定とその特徴を捉えた抽出が必要となる.

そこで本研究では, ソフトウェア保守者の観点から, デバッグ, 機能追加などのプログラムテキストの修正を行う際に, 同様の修正を行うべき箇所を発見するための利用を目的としたコードクローン情報の検索ツールの試作を行った. 本システムではユーザが入力したコード

片に対してそのコードクローンを含むファイルを検索し, 発見したコードクローンの位置をソースコード上において確認できるようにした.

また, 実際に日本語入力システム「かんな」の 2 バージョン間における修正例を用いて擬似的なデバッグを行うとして, 修正を行うべき箇所の検出を本システムを用いた場合と,grep を使用した場合の比較を定量的に行うことで本システムの有用性を示した.

以降,2 節ではコードクローン検出に関連した研究について述べ, 3 節では本システムの設計, 実装に関して説明する. さらに 4 節において本システムの性能, および本システムを実際に適用した事例に基づいて分析と考察を行う. 最後に 5 節では, 本研究のまとめと今後の課題について述べる.

2 コードクローン検出

2.1 コードクローンと既存のコードクローン検出法

コードクローンとは, ソースコード中に含まれる同一もしくは類似したコードのことであり, いわゆる “重複したコード” のことである.

コードクローンがソフトウェアの中に作りこまれる, もしくは発生する原因として次のようなものがある [8][15].

既存コードのコピーとペーストによる再利用

近年のソフトウェア設計手法を利用すれば, 構造化や再利用可能な設計が可能である. しかしながら, ゼロからコードを書くよりも既存コードをコピーして部分的な変更を加える方が信頼性が高いという反面, 実際には, コピーとペーストによる場当たり的な既存コードの再利用が多く存在する.

コーディングスタイル

規則的に何度も必要なコードはスタイルとして同じように記述される場合がある. 例えば, ユーザインターフェース処理を記述するコードなどである.

定型処理

定義上簡単で頻繁に用いられる処理. 例えば, 給与税の計算や, キューの挿入処理, データ構造アクセス処理などである.

プログラミング言語に適切な機能の欠如

抽象データ型や, ローカル変数を用いられない場合には, 同じようなアルゴリズムを持った処理を繰り返し書かなくてはならない場合がある.

パフォーマンス改善

リアルタイムシステムなど時間制約のあるシステムにおいて, インライン展開などの機能が提供されていなければ, 意図的に繰り返し書くことによってパフォーマンスの改善を図る場合がある.

コード生成ツールの生成コード

コード生成ツールにおいて, 類似した処理を目的としたコードの生成には, 識別子名等の違いはあろうとも, あらかじめ決められたコードをベースにして自動的に生成されるため, 類似したコードが生成される.

偶然

単純に偶然一致してしまう場合であるが, 大きなコードクローンになる可能性は低い.

もしコードクローンが存在した場合には, 一般的にコードの変更等が困難であると言われ, 保守容易性低下の一因となっている. このようなコードクローンによる問題に対処する方法としては,

- コードクローン情報の文書化を行うことで変更の一貫性を保つ,
- コードクローンを自動的で検出する

の 2 つがある [14]. しかしながら, コードクローン情報の文書化には全てのコードクローンに対する情報を常に最新に保つことに非常に手間がかかるため, 現実的に困難である. そこで, これまでにさまざまなコードクローン検出手法やツールが提案されている [1][2][3][4][5][6][7][8][11][15][16][17][18][19]. それぞれの手法やツールの特徴は次のようになっている (我々の開発した CCFinder は次節 2.2 参照).

Covet

[18] で定義された種々の特徴メトリクスの幾つかのメトリクス値を比較することによって, コードクローン検出を行う. 現在, 試作段階にあり, 検出対象言語は,Java である.

CloneDR[8]

抽象構文木 (AST) の節点を比較することによって, コードクローン (類似部分木) の検出を行う. また, 部分的に異なっているコードクローンも検出することが可能であり, 検出したコードクローンを自動的に等価なサブルーチンやマクロに置き換えることも可能である. 検出対象言語は,C/C++,COBOL,Java,Progress である.

Dup[2][3][4]

ユーザ定義名のパラメータ化を行った後, 行単位の比較によりコードクローンを検出する. マッチングアルゴリズムには, サフィックス木探索 [13] を用いているため線形時間で解析可能である.

Duploc[11]

前処理として, 空白やコメント等を取り除いた後, 行単位 (のハッシュ値) での表検索を用いた比較によってコードクローンを検出する. また, コードクローンの散布図等の GUI を備えたツールであり, ソースコード参照支援を行う. 検出対象言語は,C,COBOL,Python,Smalltalk である.

JPlag[19]

ソースコードを字句解析し, トークン単位での比較を行う. プログラム盗用の検出を目的として開発され, プログラム間の類似率を検出する. 検出対象言語は,C/C++,Java である.

Komondoor らの手法 [16]

関数等にまとめるのに適したコードクローンの抽出を目的として, プログラム依存グラフ (PDG) 上での各節点の比較を行うことでコードクローン (同型 (isomorphic) 部分グラフ) を検出する. 文字列比較や抽象構文木等を用いた検出方法では発見できなかったなかった非連続コードクローンや, 対応行の順番が異なるクローン, 互いに絡みあったクローン等を検出可能である. [16] で作成されたツールの検出対象言語は,C である.

Krinke の手法 [17]

AST や Traditional PDG に似た Fine-grained PDG というグラフ上での類似 (similar) 部分グラフ (同型部分グラフではない) を検出することで, コードクローンが存在すると思しき場所を検出する. [17] で作成された試作ツールの検出対象言語は,C である.

SMC[5][6][7]

まず特徴メトリクスによってソースコードをコードクローンと思しきメソッドに絞り込む. 次に絞り込まれたメソッドのペアに対し, 表検索を用いることでメソッド単位のコードクローンを検出する. 特徴メトリクスによって絞り込まれているため, 実用上ほぼ線形時間で解析可能である. また検出されたペアのメソッドは, 特徴により 18 種類に分類される. さらにそれぞれの分類については共通メソッドへの書き換え指針が示されている.

MOSS[1]

検出アルゴリズムは公開されていない. JPlag 同様, プログラム盗用の検出を目的として開発された. 検出対象言語は,Ada,C/C++,Java,Lisp,ML,Pascal,Scheme である.

いずれの手法, ツールにおいても提案者によってコードクローンの定義が微妙に異なっており, 検出されるコードクローンが異なっている. つまり, コードクローンの定義とは検出アルゴリズムそのものによって定義される. Burd ら [9] も,CloneDR,Covet,JPlag,Moss, そして我々の開発した CCFinder を含めた 5 つのツールを用いて, それぞれ検出されるコードクローンの比較が行っているが, 全ての面において他のツールよりも優れているツールはなく, 使う場面に応じて, 適切なツールを選ぶことが必要となってくると述べている.

2.2 コードクローン検出ツール CCFinder

2.2.1 概要

あるトークン列中に存在する 2 つの部分トークン列 α,β が等価であるとき, α は β のクローンであるという (その逆もクローンであるという). また,(α,β) をクローンペア (図 1 参

図 1: クローンペアとクローンクラス

照) と呼ぶ. α,β それぞれを真に包含するどのようなトークン列も等価でないとき α,β を極大クローンという. また, クローンの同値類をクローンクラス (図 1 参照) と呼び, ソースコード中でのクローンを特にコードクローンと呼ぶ.

CCFinder は, 単一または複数のファイルのソースコード中から全ての極大クローンを検出し, それをクローンペアの位置情報として出力する. CCFinder の持つ主な特徴は次の通りである.

細粒度のコードクローンを検出

字句解析を行うことにより, トークン単位でのコードクローンを検出する.

大規模ソフトウェアを実用的な時間とメモリで解析可能

例えば 10MLOC のソースコードを 68 分 (実行環境 Pentium3 650MHz RAM 1GB) で解析可能である [14].

様々なプログラミング言語に対応可能

言語依存部分を取り替えることで, 様々なプログラミング言語に対応できる. 現在は,C/C++,Java,COBOL/COBOLS,Fortran,Emacs Lisp に対応している. またプレーンテキストに対しても, 分かち書きされた文章として解析可能となっており, 未対応の言語に対しても完全一致判定によるコードクローンはとることができる.

実用的に意味を持たないコードクローンを取り除く

- コードクローンは小さくなればなるほど偶然の一致である可能性が高くなるが, 最低限一致する必要があるトークン数を指定することができるため, そのようなコードクローンを検出しないようにできる.
- モジュールの区切りを認識する

ある程度の違いは吸収可能

- ソースコード中に含まれるユーザ定義名, 定数をパラメータ化することで, その違いを吸収できる.
- クラススコープや名前空間による複雑な名前の正規化を行うことで, その違いを吸収できる.
- その他, テーブル初期化コード, 可視性キーワード (protected, public,private 等), コンパウンド・ブロックの中括弧表記等の違いも吸収することができる.

繰り返しコードに対して特別な処理を行う

例えば, コード列中で

`X A B C A B C A B C A B C Y`

のように 3 つの記号 A,B,C が繰り返していたとき, 以下の "*" が付された部分を "繰り返し" であると定義する.

`X A B C A B *C *A *B *C *A *B *C Y`

これは一見,

`X A B C *A *B *C *A *B *C *A *B *C Y`

のように定義する方が合理的であるように考えられる. しかしながら, この場合, もしコードクローンとして検出されたコード片が B C A であった場合,

`X A [B C *A] [*B *C *A] [*B *C *A] *B *C Y`

のように, このコード列中から 3 のコード片 ("[","]" で囲まれた部分) が, B C A を各要素とするクローンクラスに含まれる. すると, これら 3 つの各コード片の非繰り返しコードの長さは, 左から順に 2,0,0 と判定される.

一方,

`X A B C A B *C *A *B *C *A *B *C Y`

であれば, もしコードクローンとして検出されたコード片が B C A であっても,

`X A [B C A] [B *C *A] [*B *C *A] *B *C Y`

となり, 非繰り返しコードの長さは, 左から順に 3,1,0 と判定される. つまり, 少なくとも先頭のコード片関しては, 長さ 3 と判定可能である. もちろんコードクローン片の意味的な先頭になり得る記号が区別可能であれば, こういった問題は起こらない. しかしながら, 現実的には人間が見てもすぐにはコードクローン片の最初を判断しかねる場合もある. 例えば,

:
代入文
出力文
代入文
出力文
:

のようなコード列があった場合,2 つ目の代入文は出力文の後始末をしているのか, 次の出力に備えた処理なのか判断は難しい. ゆえに, このような定義となっている.

2.2.2 コードクローン検出処理手順

CCFinder のコードクローン検出処理は, 以下の 4 ステップで構成されている.

ステップ 1: 字句解析: ソースコードをプログラミング言語の文法に沿ってトークン列に変換する. その際, 空白とコメントは機能に影響しないので無視されるファイルが複数の場合には, 単一ファイルの解析と同じように処理できるよう, 単一のトークン列に連結する.

ステップ 2: 変換処理: 実用的に意味を持たないコードクローンを取り除くため, もしくはある程度の違いは吸収するためにトークン列を実用的に意味のあるコードクローンのみを検出するための変換を施す. 例えば, 変数名, 関数名などは全て同一のユニークなトークンに置換される.

ステップ 3: 検出処理: トークン列を比較して, 一致した部分トークン列をコードクローンとする. 但し, 指定された最小一致トークン以上の長さをもつコードクローンのみが検出される. また, トークン列の比較にはサフィックス木探索 [13] を行うため, 線形時間[1]で解析可能となる.

ステップ 4: 出力整形処理: 検出されたクローンペアについて, 元のソースコード上での位置情報を出力する.

2.2.3 検出例

実際に,CCFinder によってどのようなコードクローンが検出されるのか例を示す.

[1]構築されたサフィックス木からコードクローンを取り出すには, 木の全探索が必要であるため, トークン列の長さを n, (極大コードクローンだけではなく全ての) クローンペアの数を k をとして $O(n+k)$.

```
    1. static void foo() throws RESyntaxException
    2. {
    3.    String a[] = new String [] {"123,400", "abc"};
A1  4.    org.apache.regexp.RE pat =
A1  5.              new org.apache.regexp.RE("[0-9,]+");
A1  6.    int sum = 0;
    7.    for (int i = 0; i < a.length; ++i)
B1  8.    {
B1  9.       if (pat.match(a[i])){
B1 10.          sum += Sample.parseNumber(pat.getParen(0));}
   11.    }
C1 12.    System.out.println("sum = " + sum);
   13. }
   14. static void goo(String [] a) throws RESyntaxException
   15. {
A2 16.    RE  exp = new RE("[0-9,]+");
A2 17.    int sum = 0;
   18.    int i   = 0;
   19.    while (i < a.length)
B2 20.    {
B2 21.       if (exp.match(a[i]))
B2 22.          sum += parseNumber(exp.getParen(0));
   23.       i++;
   24.    }
C2 25.    System.out.println("sum = " + sum);
   26. }
      :
      :
```

図 2: コードクローン検出例

図 2 には, Java で互いに似通った 2 つのメソッドが書かれ, 図中の左端には行番号が付されている. ここで, 最小一致トークン数を 5 トークンに定め, 図 2 のソースコードに対しコードクローン検出を行うと, 図 2 中の `A1` (4 行目-6 行目) と `A2` (16 行目-17 行目), `B1` (8 行目-10 行目) と `B2` (20 行目-22 行目), そして `C1` (12 行目) と `C2` (25 行目) がそれぞれクローンペアとして検出される. それぞれのクローンペアの長さは順に 7, 18, 6 トークンとなっている. 見ての通り,A1 と A2 の間,B1 と B2 の間には次のような幾らかの違いが含まれているがコードクローンとして検出可能となっている.

- 名前空間の違い (e.g. “`org.apache.regexp.RE`” と “`RE`”).
- 変数名の違い (e.g. “`pat`” と “`exp`”).
- 改行とインデントの違い
- 中括弧表記の違い

これらの違いは,2.2.1 節で述べた目的のため, CCFinder のトークン変換処理によって吸収されている.

3　クローン情報検索ツール

3.1　概要

2 節において, これまでに提案された様々なコードクローン検出手法について述べたが, 分析者は, どういったコードが他と類似しており, それらがどこに存在しているのかを知るための手段としてツールを利用することが前提となっているため, 一般にコードクローン検出ツールは, grep[12] などのようにクエリやキーとなるコード片は持たず, 解析対象のソースコード中に存在する全てのコードクローンを検出する．また, 検出されたコードクローンの位置情報を視覚的に示す手法もこれまで既に幾つか提案されている [11][23] が, 同様に, 解析対象のファイル全ての組合せでの比較結果が示されることが前提となっている.

しかし, 開発・保守の過程において作成もしくは修正の検討を行っているコード片に類似している部分を探し出したいときには, それらの情報は不必要なものが大部分を占め, 余分な分析コストがかかる.

例えば, あるコード片に対するコードクローンを見つけたい場合として,

- あるコード片に対して, 変更を行い, 同じような処理を行っている部分についても同じ変更を施したい場合.
- 処理を記述したときにその前後に記述する処理の参考に出来るようなコード片が欲しい場合

などが考えられる.

そこで本研究では, 特定のコード片に対する類似箇所のみを全て抽出するシステムの試作を行う. 既存のコードクローン検出ツールを利用することで grep などのように単なる文字列一致の検出とは異なる．また, 本システムにおいては次のようなコンセプトをもって試作を行った.

- インターフェースをわかりやすく設定する.
- ユーザが入力したコード片に対してコードクローン情報を検索し, その結果を提示する.
- 検索結果をソースコード上で確認できるようにする.

3.2 概要設計

システム構成の概略図を図 3 に示す．本システムは解析対象であるファイル群と, 検索対象であるコード片に対して CCFinder を内部的に実行させ, そこから得られた結果を整理し, 表示するものである.

ユーザは,GUI 上から CCFinder に与えるオプション, 検索対象となるファイル群のリストが記述してあるファイル名および解析対象とするコード片を設定した後, 解析を実行する. 解析が終了すると, 入力したコード片のコードクローン情報を表示する.

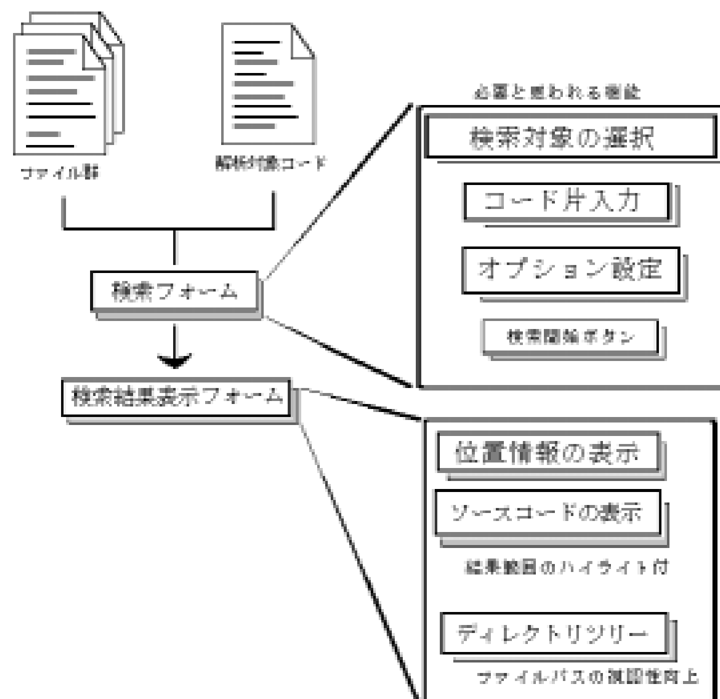

図 3: システム概観

3.3　機能設計

以降では, 図 3 に示した各機能とそのインターフェースについて説明する.

3.3.1　検索フォーム

CCFinder を実行させるときに必要なものとして, 解析対象となるファイルのリストと CCFinder の実行時オプションがある. これらの入力を検索フォーム上においてユーザに求めるものとする.

CCFinder を実行するにあたって必要な CCFinder の実行時オプションについては, ユーザーに選択させる必要があると判断したものについてのみを表示する.

検索対象ファイル群の入力

CCFinder は, 解析対象となるファイルのファイルパスが列挙されているリストファイルを入力とする. 本システムにおいては既に存在するリストファイルを選択して与える方法と, ディレクトリツリーを用いて, 選択されたファイルもしくは選択されたフォルダ以下に存在するファイルで構成するリストファイルを作成し与える方法の二つを用意する.

オプション設定

CCFinder が備えているオプションのうち, ユーザに選択させる必要があると思われるものを検索フォームにおいた. ここで,CCFinder で設定できる主なオプションとしては次のものがある.

- 解析対象言語の指定
  (C/C++,Java,COBOL, 平文テキスト等)
- 最小一致トークン数
  (一致したトークン数がこれより少ないクローンは検出しない)
- 検出するクローンペア間の関係
- 解析結果出力ファイルの指定
- 使用メモリ容量の指定

これらのうち, 本システムの利用目的を考慮したうえで, ユーザに指定してもらうものとして, 解析対象言語の指定と最小一致トークン数を選んだ. 共にユーザが求めたい検索結果

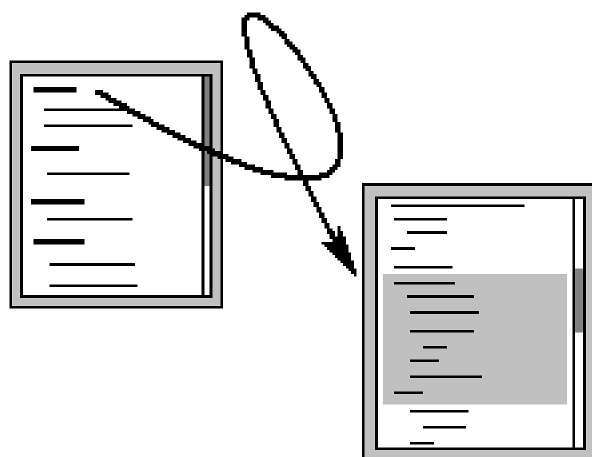

図 4: 結果表示イメージ

がこれらの値を自分で設定することで導きやすくなると思われるからである. 残りのものについてはシステム内で適当なものに決定しており, ユーザには見せないものとする.

ところで, 検出するクローンペア間の関係であるが CCFinder では "同一ファイル内","同一グループ内の相異なるファイル間", "相異なるグループ間" の 3 つの関係をそれぞれオンオフで設定できる. グループはリストファイル上で, 区切り記号を置くことで分けることが出来る. 本システムにおいてはコード片として与えられたものとそれ以外でグループ分けし, "同一ファイル内" と "相異なるグループ間" の二つをオンに設定することで, 検索対象のファイル同士での検索を実行しないようにしている.

3.3.2 検索結果出力表示フォーム

まず,CCFinder が返したクローンに関する情報を存在するファイルごとに整理し, 出力する. 解析結果を出力する際に必要であると思われる情報として, この際, クローンが発見されたファイル名, コード片およびファイルにおける発見されたクローンの位置, クローンと認識されたコードのサイズがあると思われるのでこれを表示する.

次に視覚的にクローンの見つかったファイルのパスを表現する方法として, ディレクトリツリーを用いる. また, クローンの情報をテキスト形式で表示した際, (図 4) さらに, そこからファイルを選択することで, そのソースコード上における具体的にどのコード片がクローンと認識されたのかハイライト表示などを用いて, 視覚的に分かるようにする.

図 5: 操作例

3.4 実装

本システムは Java を実装言語とし,JDK1.4 を用いて実装を行った. コードサイズは, 全ファイルで行になった. 実行に伴うシステム全体の構成を図 5 に示す. 実装の対象外として CCFinder が存在する.

二重枠で囲まれたものが, 本システムにおける主な処理であり, そこから引かれている細い矢印の先がユーザへ情報を提示する GUI である. 各矢印は情報の伝達および実行状態の遷移を表している. また,CCFinder はコードクローン検出処理内において内部的に実行される.

以降, 実行画面を交えながら本システムの実装状態および使用法について説明する. 実行画面の画像には本システムを JDK1.4 のソースコードへ適用したものを用いた.

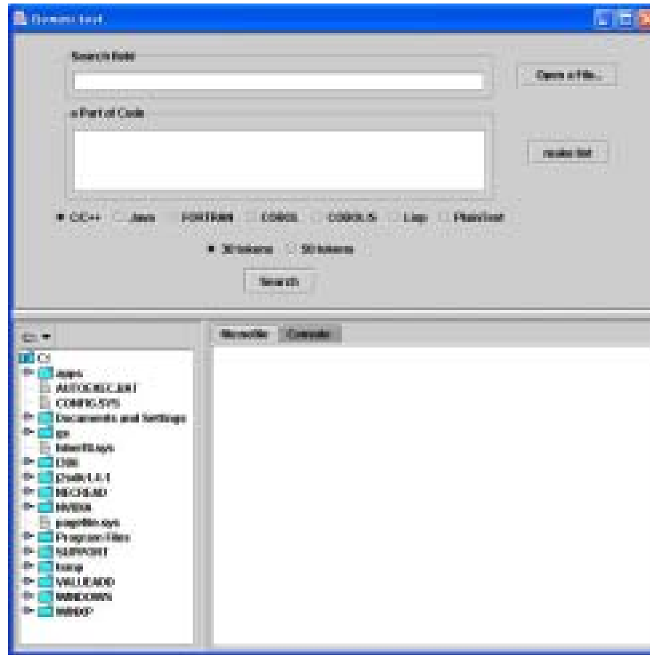

図 6: 起動直後の画面

3.4.1 検索フォーム

本システムを起動すると図 6 のような検索フォームが表示される．まず,[Search field] 欄に検索対象となるファイル群のリストが記述されたファイルのファイル名を記入する．このとき, このリストファイルを既に作成し保持しているならば, [Open a File ..] ボタンからファイルチューサーを用いて記入することができる．また, リストの作成を支援するものとして, この検索フォームの左下部にあるディレクトリツリーを使うことが出来る．このディレクトリツリーで, 検索したいフォルダやファイルを選択し, [make list] ボタンを押すと, 選択したファイルもしくは選択したフォルダ以下に存在するファイルうちで拡張子から今, 指定してある解析対象言語であると判断したファイルで構成するリストファイルを作成することが出来る．リストファイルの例として JDK1.4 のリストを図 7 に置く．

次に,[a Part of Code] 欄に解析対象とするコード片を入力し, [Search] ボタンを押すと指定してある解析対象言語および最小一致トークン数で解析を開始する．

3.4.2 検索結果の表示

解析が終了すると結果表示用にウィンドウが開き, 解析結果を表示する (図 8)．同時にディレクトリツリー上においてもクローンが発見されたファイル, もしくはそのファイルをフォル

```
D:\src\com\sun\corba\se\ActivationIDL\Activator.java
D:\src\com\sun\corba\se\ActivationIDL\ActivatorHelper.java
D:\src\com\sun\corba\se\ActivationIDL\ActivatorHolder.java
D:\src\com\sun\corba\se\ActivationIDL\ActivatorOperations.java
D:\src\com\sun\corba\se\ActivationIDL\BadServerDefinition.java
D:\src\com\sun\corba\se\ActivationIDL\BadServerDefinitionHelper.java
D:\src\com\sun\corba\se\ActivationIDL\BadServerDefinitionHolder.java
D:\src\com\sun\corba\se\ActivationIDL\EndPointInfo.java
D:\src\com\sun\corba\se\ActivationIDL\EndPointInfoHelper.java
D:\src\com\sun\corba\se\ActivationIDL\EndPointInfoHolder.java
D:\src\com\sun\corba\se\ActivationIDL\EndpointInfoListHelper.java
D:\src\com\sun\corba\se\ActivationIDL\EndpointInfoListHolder.java
D:\src\com\sun\corba\se\ActivationIDL\IIOP_CLEAR_TEXT.java
D:\src\com\sun\corba\se\ActivationIDL\InitialNameService.java
D:\src\com\sun\corba\se\ActivationIDL\InitialNameServiceHelper.java
D:\src\com\sun\corba\se\ActivationIDL\InitialNameServiceHolder.java
:
:
:
D:\src\sunw\util\EventListener.java
D:\src\sunw\util\EventObject.java
-ns
D:\TMP32162.tmp
```

図 7: ファイルリストの例

ダ内に持つフォルダのアイコンを, 赤い旗を含むものに変えることで, ディレクトリツリー上のどのファイル内においてクローンが検出されたのかを視覚的に分かるようにした (図 10).

新しく開いた結果を表示するウィンドウには,

- クローンが発見されたファイルの名前
- クローンの始まりの行と終わりの行
- 入力コード片におけるクローンの始まりの行と終わりの行
- クローンと判断された部分のトークン数

が表示される. 表示された解析結果のファイル名をクリックすると, 検出されたクローンの位置をハイライト表示されたソースコードを表示する (図 9).

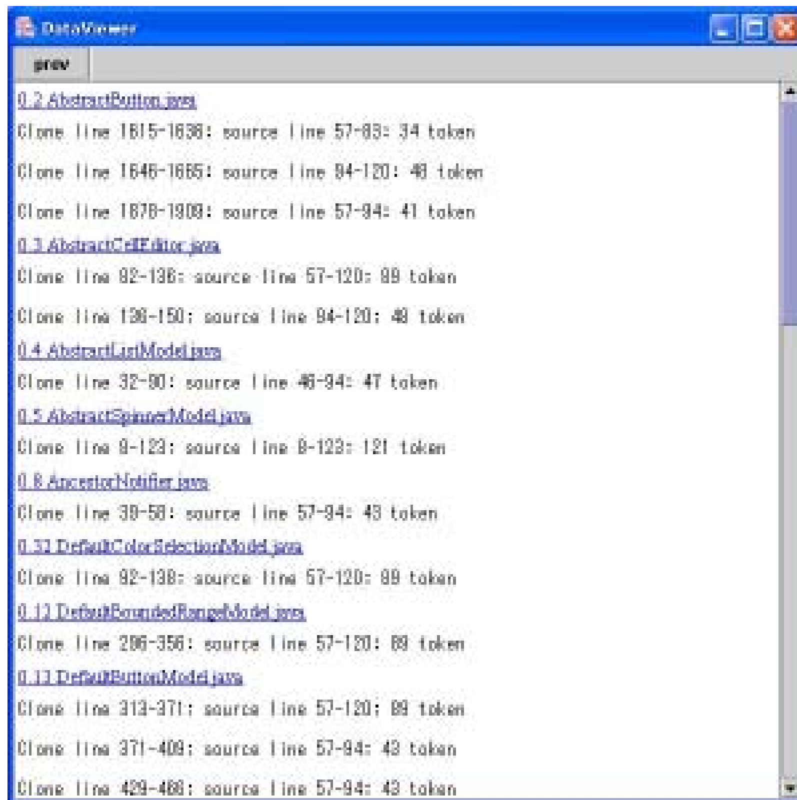

図 8: 結果出力 1

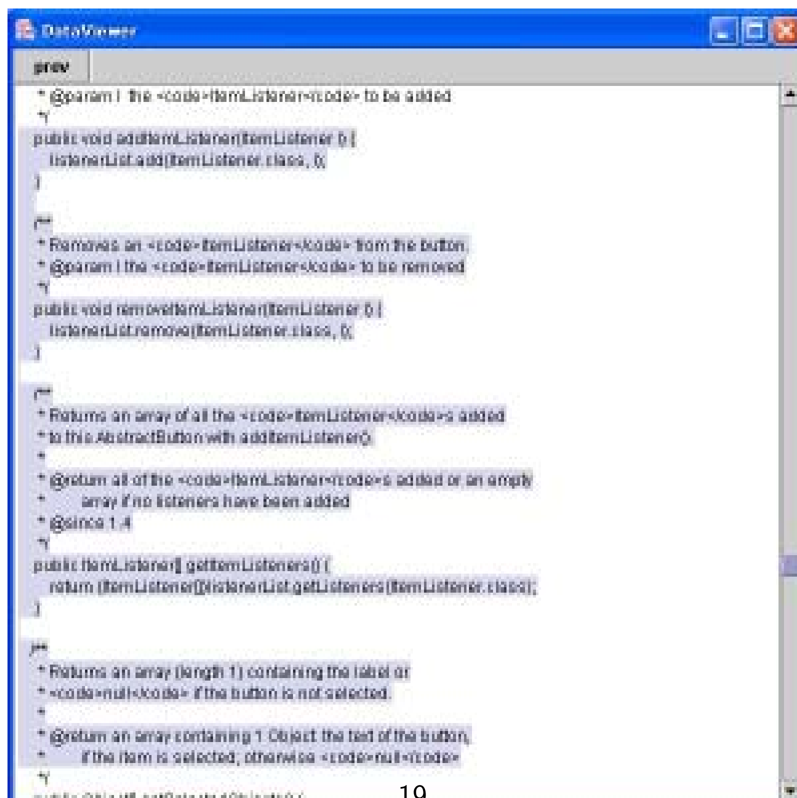

図 9: ソースコードのハイライト

図 10: ディレクトリツリーでの結果表示

4　適用

4.1　実行時性能

ここで, 実際に実装したシステムの実行時性能を確認する．この確認には JDK1.4 のソースコードを用いた．JDK1.4 のソースコードのサイズその他を表 1 に示す.

このソースコードを処理するのににかかった処理時間を表 2 に示す. 実行環境は,CPU は Pentium4 2.00GHz, 主記憶容量は 512MB, OS は WindowsXP/HomeEdition SP1,VM は JDK1.4 付属 VM である．また, 最小一致トークン数を 30 トークンと設定した.

解析の処理には, 結果の表示処理に関する時間も含まれる．結果の表示までに約 2 分かかった．しかし, コードの編集時などにインクリメンタルな処理を行うには, 検索対象の選択のしかたなどに工夫することが必要である.

4.2　「かんな」への適用

オープンソース・ソフトウェア開発サイト SourceForge.jp[20] において開発されている日本語入力システム「かんな」[10] で実際に行われたセキュリティー問題の修正履歴を用いた擬似的なデバッグを本システムの “修正への適用例” として挙げる．なお,「かんな」の開発言語は C 言語である.

4.2.1　修正に関するデータ

日本語入力システム「かんな」の Version3.6,Version3.6p1 間でのセキュリティー問題の修正において, バッファのオーバーフローを検出するコードが挿入された．この修正においては, 全部で 3 ファイル,30 箇所に対して修正が施してあるが, うち 1 ファイル,21 箇所についてはバッファを仲介した処理を行っており, その際に発生する可能性のあるオーバーフローの検出を行うコードがこの修正によって追加された．この 21 箇所では, 変数名が少しだけ異なる変数に対してほぼ共通した処理を主として行っている (図 11 参照).

図 11 のコードに対して, この修正を行なった結果図 12 のようにコードを追加された.

表 1: 処理対象データサイズ

コードサイズ (loc)	約 118 万行
ファイル数	3391 ファイル
対象コード片	32 行
発見クローン数	40

表 2: JDK1.4 の処理時間

処理内容	処理時間
起動	5 秒
ディレクトリツリーからのリストの作成	一秒程度
コードクローン解析	約 2 分
ソースコードの表示	1 秒以内

```
1:   ir_debug( Dmsg(10, "ProcWideReq3 start!!\n") );
2:
3:   buf += HEADER_SIZE;  Request.type3.context = S2TOS(buf);
4:   buf += SIZEOFINT;    Request.type3.buflen = S2TOS(buf);
5:
6:   ir_debug( Dmsg(10, "req->context = %d\n", Recuest.type3.context) );
7:
```

図 11: 修正前の典型的なコード

そこで, この例として挙げたコード以外の 20 箇所を見つけ出すために,

コード片 1

```
3:   buf += HEADER_SIZE;  Request.type3.context = S2TOS(buf);
4:   buf += SIZEOFINT;    Request.type3.buflen = S2TOS(buf);
```

コード片 2

```
1:   ir_debug( Dmsg(10, "ProcWideReq3 start!!\n") );
2:
3:   buf += HEADER_SIZE;  Request.type3.context = S2TOS(buf);
```

コード片 3

```
3:   buf += HEADER_SIZE;  Request.type3.context = S2TOS(buf);
```

の 3 種類のコード片を, それぞれ「かんな」Version3.6 の全ソースコード (96 ファイル, 約 8 万 8 千行) を検索の対象として与え, 本ツールを用いて検索を行った.

また, 本システムでのデバッグとの比較対照として grep を用いた修正位置の特定を行った. その際,grep 検索の引数としては, バッファ処理時に使われる変数名の共有部分が最大となる "Request.type" を用いた.

```
 1:   ir_debug( Dmsg(10, "ProcWideReq3 start!!\n") );
 2:
+3:   if (Request.type3.datalen != SIZEOFSHORT * 2)
+4:     return( -1 );
+5:
 6:   buf += HEADER_SIZE;  Request.type3.context = S2TOS(buf);
 7:   buf += SIZEOFINT;    Request.type3.buflen = S2TOS(buf);
 8:
 9:   ir_debug( Dmsg(10, "req->context = %d\n", Recuest.type3.context) );
10:
```

図 12: 修正後の典型的なコード

4.2.2 結果とその分析

今回行った検索の結果は, 表 3 のとおりである.

表 3: 本システムによる検索結果

検索対象コード片	コード片 1	コード片 2	コード片 3
総修正箇所	30	30	30
類似修正箇所	21	21	21
検出されたクローンの数	25	17	42
結果から発見できる修正箇所	15	17	18
発見されなかった修正箇所	6	4	3
修正箇所と関係がないクローン	0	0	0

検出したクローンの数が修正箇所よりも多いのはバッファに関する処理が例としてあげたコードよりも繰り返し行われていたために, 同じ修正箇所の周辺で複数検出された場所があるためである. また,3 つの結果のどれも修正された箇所以外は検出しなかった.

これら 3 つの方法で検出されなかったものについては, 本ツールに与えたコード片に対して

- 修正された箇所のコードが短い
- 修正された箇所では対象したコード片を構成する文が連続していなかった

などの理由であった.

次に,grep による検索の結果は, 表 4 のとおりである.

検索されなかった一箇所は, 図 11 の 1 行目の処理のみを行う関数であった.

この2つの結果をf値を用いて比較するf値とは完全性と効率性から検索結果の精度を評価するものであり,

$$f\text{ 値} = \frac{2 \times \text{完全性} \times \text{効率性}}{\text{完全性} + \text{効率性}}$$

で求められる.

ここでは

- 完全性とは必要な情報のうち実際に検索された情報の割合
- 効率性とは実際に検索された情報のうち必要な情報の割合

と定義する.

f値の値が大きいほど, 情報検索の精度が高いとされる.

f値を計算した結果は表5のようになる. 本システムの検索結果としてはコード片2で検索したものを代表として与える.

これから, 本システムは必ずしも全ての修正箇所を提示することが出来るわけではないが, 利用しない検索結果が出ないぶんだけgrep検索よりも本システムを用いた検索のほうがデバッグに有用であることが分かる.

表4: grep検索の結果

	grep
総修正箇所	30
類似修正箇所	21
検出した行数	243
発見した修正箇所	20
発見されなかった修正箇所	1
修正箇所と関係のない箇所	38

5　まとめと今後の課題

本研究では開発・保守工程におけるコード修正作業等を支援するため，コードクローン検出ツール CCFinder を利用した, 特定コード片に対するコードクローン情報検索システムを試作した.

本システムにおいては，特に，あるコード片に対して修正を行う際に，類似した処理を行っている他のコードにも同様の修正を行うかどうかの検討を行いたい場合，もしくは，ある処理を記述している際やバグへの対処を行っている際に，類似コードから処理方法のヒントやバグへの対応策を得たい場合を想定している．そのため，コードクローン検出の際に，ユーザに対し特定コード片の指定を可能にし，解析結果として当該コード片のみに対するコードクローンの位置情報や実際のコードをユーザに示す．また，コードクローンが存在するファイルまたはディレクトリをディレクトリツリー上に示すことで，ユーザはファイルシステム内でどのようにコードクローンが分散しているのかを直観的に認識することができる.

さらに，本システムを日本語入力システム「かんな」のソースコードに対して適用することで，その有効性を確認した．本適用においては,「かんな」の開発において，実際に行われた修正履歴を用いて本システムを使った擬似的なデバッグを行った．特に，ある 2 バージョン間におけるバッファオーバーフローが起こる問題に対して，類似箇所に対して同様の修正が行われた箇所をどの程度発見することができるのかを，本システムを用いた場合と grep を用いた場合で比較を行った．これら 2 つの場合を，f 値を用いて完全性と効率性から検索結果の精度の評価したところ，本システムが高い精度をもつことが確認され，我々が想定したコード修正作業の支援において有用であることがであることを示した.

今後の課題としては

- 検出, 分析, 再構築間のサイクルをインクリメンタルに行うことによる実用性の向上,
- クローンペア, クローンクラスの優先順位付け分類した表示,
- コード作成時にクローンを表示することによるコード作成容易性の向上,

表 5: f 値の計算表

	完全性	効率性	f 値
grep 検索	95%	34%	0.50
本システム	81%	100%	0.90

- 実際の開発現場での適用実験

などが挙げられる．

謝辞

本研究において, 常に適切な御指導および御助言を頂きました 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 井上 克郎 教授に深く感謝致します.

本研究において, 常に適切な御指導および御助言を頂きました 同 楠本 真二 助教授に深く感謝致します.

本研究において, 常に適切な御指導および御助言を頂きました 同 松下 誠 助手に深く感謝致します.

本研究において, 逐次適切な御指導および御助言を頂きました 科学技術振興事業団 さきがけ研究 21 神谷 年洋 研究員に深く感謝致します.

本研究において, 逐次適切な御指導および御助言を頂きました 同博士前期課程 2 年 植田泰士氏に深く感謝致します.

最後に, その他様々な御指導, 御助言を頂いた大阪大学 大学院情報科学研究科 コンピュータサイエンス専攻 井上研究室の皆様に深く感謝いたします.

参考文献

[1] A. Aiken, "A System for Detecting Software Plagiarism (Moss Homepage)", `http://www.cs.berkeley.edu/~aiken/moss.html` [Last visited 1st Feb. 2003].

[2] B.S. Baker, "A Program for Identifying Duplicated Code", *Computing Science and Statistics*, 1992, 24:49-57.

[3] B.S. Baker, "On Finding Duplication and Near-Duplication in Large Software Systems", *Proceedings the 2nd Working Conference on Reverse Engineering*, 1995, 86-95.

[4] B.S. Baker, "Parameterized Duplication in Strings: Algorithms and an Application to Software Maintenance", *SIAM Journal on Computing*, 1997, 26(5):1343-1362.

[5] M. Balazinska , E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis, "Advanced Clone-Analysis to Support Object-Oriented System Refactoring", *Proceedings the 7th Working Conference on Reverse Engineering*, 2000, 98-107.

[6] M. Balazinska , E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis, "Measuring Clone Based Reengineering Opportunities", *Proceedings 6th IEEE International Symposium on Software Metrics*, 1999, 292-303.

[7] M. Balazinska , E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis, "Partial redesign of Java software systems based on clone analysis", *Proceedings 6th IEEE International Working Conference on Reverse Engineering*, 1999, 326-336.

[8] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone Detection Using Abstract Syntax Trees", *Proceedings IEEE International Conference on Software Maintenance-1998*, 1998, 368-377.

[9] E. Burd, and J. Bailey, "Evaluating Clone Detection Tools for Use during Preventative Maintenance", *Proceedings 2nd IEEE International Workshop on Source Code Analysis and Manipulation*, 2002, 36-43.

[10] 日本語入力システム「かんな」`http://canna.sourceforge.jp/` [Last visited 19 Feb. 2003]

[11] S. Ducasse , M. Rieger, and S. Demeyer. "A Language Independent Approach for Detecting Duplicated Code", *Proceedings IEEE International Conference on Software Maintenance-1999*, 1999, 109-118.

[12] grep `http://www.gnu.org/software/grep/grep.html` [Last Visited 19 Feb. 2003]

[13] D. Gusfield, *Algorithms on Strings, Trees, And Sequences*, Cambridge University Press, 1997.

[14] 井上克郎, 神谷年洋, 楠本真二, "コードクローン検出法", **コンピュータソフトウェア**, 2001, 18(5):47-54.

[15] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code", *IEEE Transactions on Software Engineering*, 2002, 28(7):654-670.

[16] R. Komondoor, and S. Horwitz, "Using slicing to identify duplication in source code", *Proceedings 8th International Symposium on Static Analysis*, 2001.

[17] J. Krinke, "Identifying Similar Code with Program Dependence Graphs", *Proceedings 8th Working Conference on Reverse Engineering*, 2001, 562-584.

[18] J. Mayland, C. Leblanc, and E. Merlo, "Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics", *Proceedings IEEE International Conference on Software Maintenance-1996*, 1996, 244-253.

[19] L. Prechelt, G. Malpohl, and M. Philippsen, "Finding plagiarisms among a set of programs with JPlag", *resubmitted to Journal of Universal Computer Science*, 2001. `http://www.ipd.uka.de/~prechelt/Biblio/#jplag` [Last visited 1 Feb. 2002].

[20] SourceForge.jp `http://sourceforge.jp/` [Last visited 19 Feb. 2003]

[21] 植田泰士, 神谷年洋, 楠本真二, 井上克郎, "クローン検出ツールを用いたソースコード分析ツールの試作", **電子情報通信学会技術研究報告** *SS2001-14*, 2001, 101(240):17-24.

[22] Y. Ueda, Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, "Gemini: Code Clone Analysis Tool", *Proceedings 1st International Symposium on Empirical Software Engineering*, 2002, 2:31-32.

[23] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue, "Gemini: Maintenance Support Environment Based on Code Clone Analysis", *Proceedings 8th IEEE International Symposium on Software Metrics*, 2002, 67-76.