保守支援を目的としたコードクローン情報検索ツールの試作
Code Clone Search Tool for Software Modification

泉田 聡介 † **植田 泰士** † **神谷 年洋** § **楠本 真二** † **井上 克郎** †
Sousuke Izumida Yasushi Ueda Toshihiro Kamiya Shinji Kusumoto Katsuro Inoue

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 [3]. 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. もしあるコード片にバグ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, 機能追加においても同様である. そこで, 大規模ソフトウェアから効率よくコードクローンを検出する手法が求められている.

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 [1][2][3][5][7][8]. 我々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[6]) を開発してきており, 様々なソフトウェアに対する適用を行ってきている. これらの適用の中で, CCFinder は大規模なソースコードも, 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 の出力結果は, テキスト情報であり, 実際の保守作業において利用する場合には, コードクローンの位置情報を直感的に把握することが困難であった.

本研究では, コードクローン検出ツール CCFinder で検出されたコードクローン情報に基づいて, ソフトウェア保守を効率よく実施するためのコードクローン検索ツールについて提案する. また, 本ツールの有効性を grep との比較を用いて示した.

2. CCFinder

2.1 コードクローン

ある系列中に存在する 2 つの部分系列 α, β が「等価」であるとき, $C(\alpha, \beta)$ と書き, α は β のクローンであると言う. また, α, β の組をクローンペアと呼ぶ. 通常, C は, 反射, 推移, 対称律を満たし, 同値関係である. α を含む同値類を α のクローンクラスと言う.

任意の α, β に対して $C(\alpha, \beta)$ ならば, それぞれの部分系列 α', β'(α'$< \bullet\alpha$, β'$< \bullet\beta$ と書く) に対して $C(\alpha$',β') が成り立つ. また, 任意の α", β"($\alpha < \bullet\alpha$", $\beta < \bullet\beta$") に対して $C(\alpha$",β") ではなく, かつ $C(\alpha, \beta)$ ならば, α, β を極大クローンペアと呼ぶ. 本稿では, ある部分系列 α に対し, 別の部分系列 β が α と極大クローンペアを構成するとき, α を単に「クローン」と呼ぶ.

†大阪大学 大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University
§科学技術振興事業団 さきがけ研究 21
PRESTO, Japan Science and Technology Corp

系列 S が与えられたとき, S 中の極大クローンペア α, β (ただし $\alpha \neq \beta$) を全て発見することを, クローン発見問題と言う. 通常, 「クローン検出」あるいは「重複コード発見」ツールと呼ばれるものは, このクローン発見問題を解くことを目的としている. ただし, ある一定の長さ以上の極大クローンペアのみを出力するようにしているのが普通である. 短いクローンは, 多数発見されることが多いが, その意味や存在には, 興味のない場合が多いからである.

2.2 コードクローン検出ツール CCFinder

コードクローン検出ツール 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コードクローンを検出し, それをクローンペアの位置情報として出力する. CCFinder の持つ主な特徴は以下の通りである.

(1) ソースコードをトークン単位で直接比較することによりクローンを検出する.

(2) クローン検出アルゴリズムにサフィックス木を用いることで高速化を図っており, 数百万行規模のシステムにも実用時間で解析可能である.

(3) 実用的に意味のないクローン (モジュールの先頭にあるテーブルの初期化文等の繰返し) は検出しない.

(4) 複数のプログラミング言語 (C/C++, JAVA, COBOL, Fortran, EmacsLisp, Plain text) への対応も実現している.

図 1 に CCFinder の出力例を示す. CCFinder の出力はテキストファイルであり, その内容は, 最初の部分が検索時のオプション, 次の file description の部分で入力となったソースファイルに振られた内部的な ID 番号, 最後の clone の部分で検出されたクローンペアそれぞれについての開始位置, 終了位置となっている. 例えば, 図 1 の下線部分はファイル ID0.95 のファイルの 2331 行目の 5 カラム目から 2333 行目の 5 カラム目までと, ファイル ID1.0 のファイルの 1 行目 1 カラム目から 3 行目 1 カラム目までがコードクローンであることを意味している.

このように, クローンの検出結果はクローンペアの位置情報のみであるので, ユーザの利用目的に応じたユーザインタフェースを作成する必要がある.

3. コードクローン検索ツール

3.1 基本方針

今回作成したコードクローン検索ツールは, デバッグ時の利用と機能追加時の利用を想定している. 具体的には, 修正 (あるいは, 機能追加) 対象のコード片が特定された時に, そのコード片に対応するコードクローンを前もって検出しておき, それらについても修正 (あるい

```
#version:  ccfinder 4.5h
#langspec:  C
#option:  -b 12,1
#option:  -k:  -
#option:  -r abdfikmnpstv
#option:  -c w-f-g
#beginfile description
0.0 1475 3441 D:\Cannna_source\Canna36\canuum\canna.c
0.1 3213 9753 D:\Cannna_source\Canna36\ccustom\ccustom.c
0.2 3584 11442 D:\Cannna_source\Canna36\ccustom\lisp.c
0.3 588 1600 D:\Cannna_source\Canna36\ccustom\parse.c
          ⋮
          ⋮
#endfile description
#beginsyntax error
#endsyntax error
#beginclone
0.95 2331,5,7202 2333,5,7216 1.0 1,1,0 3,1,14 14
0.95 2348,5,7258 2350,5,7272 1.0 1,1,0 3,1,14 14
0.95 2349,5,7265 2351,5,7279 1.0 1,1,0 3,1,14 14
0.95 2350,5,7272 2352,5,7286 1.0 1,1,0 3,1,14 14
0.95 2373,5,7383 2375,5,7397 1.0 1,1,0 3,1,14 14
0.95 2374,5,7390 2376,5,7404 1.0 1,1,0 3,1,14 14
          ⋮
          ⋮
#endclone
```

図 1: CCFinder の出力例

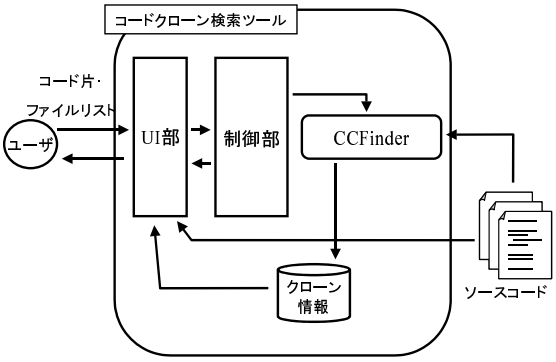

図 2: システム構成

は, 機能追加）が必要であるかを検討するための支援を行う.

コードクローンの検出部分として, CCFinder を用いているが, 検索ツールとしては, 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, ユーザに対してわかりやすく提示する必要がある. また, それらの処理を簡便に行えることも重要である.

3.2 概要

開発した検索ツールでは, まず, 入力として, ユーザが指定するコード片とコードクローン検出対象となるファイル名のリストを与える. その結果, 出力としてユーザが指定したコード片のクローンを含むファイルのリストを表示する.

検索ツールの構成図を図 2 に示す. 図のように本ツールはユーザの入力情報から, CCFinder へのオプション, 入力ファイルを作成し, 検索結果をクローンが検出されたファイル別に表示を行う.

検索ツールの画面例を図 3 に示す. A にコードクローン検出対象となるファイル名のリストを入力し, B にはユーザが指定するコード片を入力する. コード片はエディタ等からコピーとペーストによって入力することもできる. また, A へ入力するリストは C に提示されているディレクトリ構造から検索対象ファイルを選ぶ操作によっても作成できる.

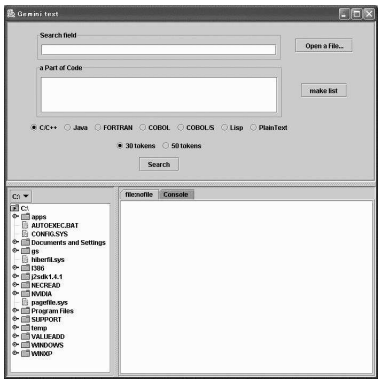

図 3: 開始画面

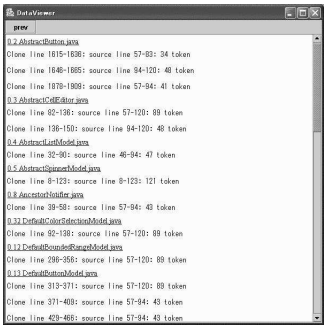

図 4: 検索結果 (1)

コードクローンの検索が終了すると, 結果を表示するための新たなウィンドウが開かれ (図 4), ユーザが指定したコード片のクローンを含むファイルのリストが, 新しいウィンドウに表示される. ユーザはこのリストからファイルを選択して (図中の下線のついたファイル名をクリックする), そのファイルのテキストを表示させ, コードクローンの位置を確認することができる (検索されたコードクローンはハイライトで表示される)(図 5).

3.3 適用例

開発したツールを用いた疑似デバッグによって, ツールの適用例を示すとともに, その有効性を評価する.

実験では, SourceForge で開発されている日本語入力システム「かんな」[4] の修正例を用いた. 具体的には, 「かんな」バージョン 3.6 とバージョン 3.6p1 間でのセキュリティ問題の修正で, バッファ処理の前にオーバーフローを調べる処理を追加してあり, その中でほぼ同じ修正を行っている 21 箇所を対象とした.

修正されたコード片の一つから残り 20 個のコード片を検索する作業において, 標準的な検索ツールである grep と本ツールを比較する. まず, grep を用いた検索では, 修正箇所で使用されている “Request.type” という変数

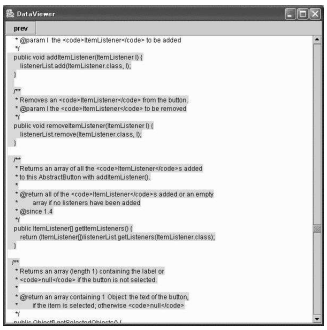

図 5: 検索結果 (2)

表 1: 評価結果

	検索結果	検索時間
grep	243 行 (58ヵ所)	1 秒以内
本ツール	17ヵ所	8 秒

名の一部で検索を行った. 本ツールではバッファ処理を行っている部分の 2 行を入力コード片として与えて検索を行った. 検索対象ファイルは「かんな」ver3.6 の全ソースコード (C 言語, 約 21, 000 行) である.

検索結果と検索に要した時間を表 1 に示す. grep での検索結果では 234 行が検出された. このうち, 2 行以上連続したコード片を 1 つの修正対象コード片とみなすと, 全部で 58 箇所となった. このうち, 134 行 (20 箇所) が修正箇所に関連した場所であった. 一方, 本ツールの検索結果では, 17 箇所が検出され, 4 箇所が発見されなかった. しかし, 発見された 17 箇所は全て正しく修正箇所を示していた. 本ツールで発見されなかった部分は, 入力コード片として与えた 2 行が連続していなかった部分であった.

2 つのツールの検出結果を f 値 [9] を用いて比較する. f 値とは完全性と効率性から情報検索の精度を評価するものであり,

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

と定義されている.

ここで, 完全性は必要な情報のうち実際に検索された情報の割合, 効率性は実際に検索された情報のうち必要な情報の割合であり, 両方ともに高いほど優れた検索システムであると判断される. 従って, f 値が大きいほど, 情報検索の精度が高いといえる.

本実験結果に対して, f 値を計算した結果を表 2 に示す. 本ツールの結果が, grep を用いた検索結果よりもよい結果を示している.

4. まとめ

本論文では, ソフトウェア保守を支援するためのコードクローン検索ツールについて述べ, 本ツールの有効性を示した. 今後の課題としては, 実際の保守現場での適用・評価や検索されたコードクローンの絞り込み等の改良が考えられる.

表 2: 検索結果の比較

	完全性	効率性	f 値
grep	95%	34%	0.50
本ツール	81%	100%	0.90

参考文献

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

[2] M. Balazinska , E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis, “Measuring Clone Based Reengineering Opportunities”, *Proceedings of the Sixth InternationalSymposium on Software Metrics (METRICS99)*, 1999, 292-303.

[3] I.D. Baxter, A. Yahin, L. Moura, M. Sant’Anna, and L. Bier, “Clone Detection Using Abstract Syntax Trees”, *Proceedings of the 14th International Confernce on Software Mintenance (ICSM98)*, 1998, 368-377.

[4] 日本語入力システム「かんな」 `http://canna.sourceforge.jp/`

[5] 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.

[6] 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.

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

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

[9] 徳永 武信, “情報検索と言語処理”, 東京大学出版会, 2002.