多様なプログラミング言語に対応可能なコードクローン検出ツール CCFinderSW*

瀬村　雄一[†a)]　吉田　則裕[††b)]　崔　恩瀞[†††c)]　井上　克郎[†d)]

CCFinderSW: Clone Detection Tool with Flexible Multilingual Tokenization*

Yuichi SEMURA[†a)], Norihiro YOSHIDA[††b)], Eunjong CHOI[†††c)], and Katsuro INOUE[†d)]

あらまし　近年実務に使用されるプログラミング言語は多様化し，ある一つのプログラミング言語においてもその文法はバージョンごとに差異をもつ．字句単位のコードクローン検出ツール CCFinderX は，多様な言語に対応するためのシンプルな仕組みをもたない．提案ツールとして，構文解析器生成系の一つである ANTLR の構文定義記述を入力として与えることで，新たな言語の字句解析が可能になるコードクローン検出ツール CCFinderSW を開発した．評価実験では，42 言語の構文定義記述からコメントや予約語，文字列リテラルの情報を抽出し，81%の言語でこれら 3 種類の情報が抽出可能であることを示した．また，C++で記述されたソースコードに対するコードクローン検出において CCFinderX と出力を比較し，ほぼ同等の検出能力をもつことを示した．

キーワード　コードクローン，字句解析，ANTLR

1. まえがき

コードクローンとは，ソースコード中に存在する互いに一致，または類似したコード片を指し，主に既存のコード片のコピーアンドペーストによって生成される [1]．一般的にコードクローンの存在は，ソフトウェア保守を困難にしている大きな要因の一つとして挙げられている．あるコード片にバグが見つかった場合，そのコード片のコードクローンにもバグが含まれる可能性が高い．そのため，開発者はあるコード片にバグが見つかった場合，そのコード片の全てのコードクローンに対して同一の修正を行うか検討する必要がある [2]．現在まで多くのコードクローン検出ツールが提案されているが [3]，近年では実務に使用されるプログラミング言語は多様化し，ある一つのプログラミング言語においても，バージョンによってその文法には差異がある．しかし，既存のコードクローン検出ツールは，多様なプログラミング言語に対応するためのシンプルな仕組みをもたない [4]．

コードクローン検出ツールの一つである CCFinderX は，C++，C#，Java，COBOL，Visual Basic の言語で記述されたソースコードからコードクローンを検出することが可能である(注1)．CCFinderX は字句単位のコードクローンを検出するための前処理として，ソースコードを言語の文法に沿って字句単位に分割している．一般的に字句解析と呼ばれるこの処理によって，ソースコードのフォーマット，コメントの有無，変数名や関数名の違いを無視した実用的で意味のあるコードクローンを検出することができる [5]〜[8]．CCFinderX は，新たな言語の字句解析部を実装することでコードクローン検出が可能になる仕組みをもつが，この字句解析部の実装は手間のかかる作業である．このような場合に，新たな言語に容易に対応するため

† 大阪大学，吹田市
Osaka University, Suita-shi, 565–0871 Japan
†† 名古屋大学，名古屋市
Nagoya University, Nagoya-shi, 464–8601 Japan
††† 京都工芸繊維大学，京都市
Kyoto Institute of Technology, Kyoto-shi, 606–8585 Japan
a) E-mail: y-semura@ist.osaka-u.ac.jp
b) E-mail: yoshida@ertl.jp
c) E-mail: echoi@kit.ac.jp
d) E-mail: inoue@ist.osaka-u.ac.jp
* 本論文は学生論文特集秀逸論文である．
DOI:10.14923/transinfj.2019PDP0025

(注1)：http://www.ccfinder.net/

のシンプルな仕組みを利用することで，ツール開発者の手間を減らすことができる [9].

本研究では，多様なプログラミング言語に容易に対応することができるコードクローン検出ツールの開発を目的として，構文解析器生成系の一つである ANTLR で利用される構文定義記述から，字句解析に必要な文法を自動的に抽出するモジュールを開発した．そしてこのモジュールが抽出した文法を用いて，言語の文法に沿ったコードクローン検出が可能な CCFinderSW を開発した．CCFinderSW の利用者は，構文定義記述が集められたリポジトリ grammars-v4(注2) から対象言語の構文定義記述を取得し，ツールの実行時に入力としてそのまま与えることでコードクローン検出を行うことができる．

また，CCFinderSW に関する三つの評価実験を行った．一つ目の実験では，ANTLR で利用できる構文定義記述を集めたリポジトリ上の 42 の構文定義記述を対象に，本研究で開発したモジュールがどの種類の文法を抽出可能かを確認した．二つ目の実験では，C++ のソースコードに対してコードクローン検出を行い，CCFinderX と CCFinderSW の検出結果の差異を分析した．最後に，Verilog HDL のソースコードに対して CCFinderSW を用いてコードクローン検出を行い，Precision と Recall を測定した．

以降，**2.** では本研究の背景としてコードクローンについて，そしてコードクローン検出ツールである CCFinderX の説明を行う．**3.** では，構文解析器生成系の一つである ANTLR の本研究での利用方法について説明し，次に構文定義記述から文法情報を抽出する手法について説明を行う．**4.** では三つの評価実験について記述し，**6.** ではまとめと今後の課題について述べる．

2. 背　　景

本章では本研究の背景として，コードクローンとその分類，コードクローンの検出技術について記述し，字句単位のコードクローン検出ツールである CCFinderX についての説明を行う．

コードクローンとは，ソースコード中に存在する互いに一致，または類似したコード片を指す．コードクローンは，主に既存のコード片のコピーアンドペーストによって生成される [1]．一般的に，互いにコードクローンとなるコード片はクローンペアと呼ばれ，クローンペアにおいて推移関係が成り立つコードクローンの集合はクローンセットと呼ばれる．またクローンペアの二つのコード片に対し，それぞれを包含するいかなる字句列も等価でないとき，極大クローンペアと呼ぶ．

コードクローンには，普遍的定義は存在しない．本論文では，コードクローンの定義として二つの分類を用いる [10]．タイプ 1 のコードクローンは，空白，タブ文字，改行やコメントなどを除いて一致するコードクローンを指す．タイプ 2 はタイプ 1 の条件に加えて，リテラル，型，識別子を除いて一致するコードクローンを指す．タイプ 2 のコードクローンを検出することで，変数名に差異があるが類似した処理を行うクローンペアをリファクタリング候補として提示したり，変数名の修正漏れを含むコードクローンを検出したりすることができる [7], [8].

CCFinderX は，C++，C#，Java，COBOL，Visual Basic といったプログラミング言語に対応した字句単位のコードクローン検出ツールであり，タイプ 1 とタイプ 2 のコードクローンを検出することができる．CCFinderX は字句単位のコードクローンを検出するための前処理として，ソースコードを言語の文法に沿って字句単位に分割している．一般的に字句解析と呼ばれるこの処理によって，ソースコードのフォーマット，コメントの有無，変数名や関数名の違いを無視した実用的で意味のあるコードクローンを検出することができる [5]〜[8].

以下に，CCFinderX を構成する四つの Step について記述する．

Step 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字句列に変換する．この際，空白とコメントは機能に影響しないので無視される．

Step 2: 変換処理

分割された字句列のうち，識別子を同一の字句に変換する．この処理はタイプ 2 のコードクローンを検出するために行われる．

Step 3: クローン検出

変換された字句列を比較し，コードクローンを検出する．比較には，suffix-tree を用いたアルゴリズムを採用している．

Step 4: 出力整形

検出されたコードクローンをクローンペアとして，

(注2)：https://github.com/antlr/grammars-v4

表 1　コードクローン検出ツールとその対応言語

ツール名	対応言語
CCFinderX	Java, C/C++, COBOL, Visual Basic, C#
DECKARD [11]	Java, C, PHP, Solidity
SourcererCC [12]	Java, C/C++, Python
Oreo [13]	Java
DeepSim [14]	Java

出現するファイル・行番号などを出力する．

CCFinderX は，新たな言語の字句解析部を実装することでコードクローン検出が可能になる仕組みをもつが，この字句解析部の実装は手間のかかる作業である．既存のコードクローン検出ツールの問題点として，多様なプログラミング言語に対応するためのシンプルな仕組みをもたないことが挙げられる．

また，CCFinderX など数多くのコードクローン検出ツールが開発されているが，表 1 に示すとおり対応プログラミング言語の数は限られる．表 1 は，CCFinderX 及び主要国際会議で発表されたコードクローン検出ツール四つとそれらの対応言語である．

3. 提案ツール：CCFinderSW

本研究では多様なプログラミング言語に対応したコードクローン検出ツールを開発することを目的として，字句単位のコードクローン検出における字句解析に必要な文法情報を，構文解析器生成系の一つである ANTLR の構文定義記述から自動的に抽出するモジュールを開発した．そしてそのモジュールを用いて，ANTLR の構文定義記述を入力として与えることで，多様な言語のコードクローン検出が可能な CCFinderSW を開発した．CCFinderSW は GitHub で公開されている(注3)．

3.1 では CCFinderSW の処理概要と各処理の詳細について記述し，**3.2** では構文定義記述解析モジュールの開発について記述する．

3.1 CCFinderSW の処理概要

図 1 は CCFinderSW の処理概要を表したものである．これは CCFinderX の処理概要に基づき，ソースコードを入力としてクローンペア情報を出力するために，字句解析，変換処理，クローン検出，出力整形を行う．本ツールは Java を用いて，一から実装を行った．構文定義記述解析モジュールは，字句単位のコードクローン検出が行う言語依存の処理であるコメント除去及び識別子変換に必要な文法を，ANTLR の構文

(注3)：https://github.com/YuichiSemura/CCFinderSW

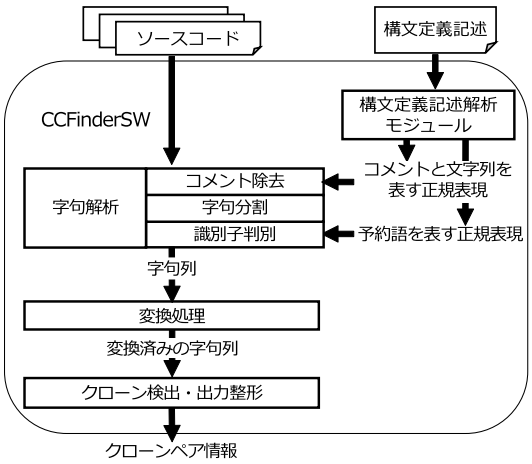

図 1　CCFinderSW の処理概要

定義記述から抽出する．具体的には，コメントや文字列リテラル，予約語の文法を抽出し，正規表現として出力する．字句単位のコードクローン検出手法は，等価な字句列をコードクローンとして検出するため，字句解析は行うが構文解析は行わない．そのため，字句解析で利用するコメントや文字列リテラル，予約語の文法のみ抽出すれば，コードクローン検出を行うことができる．字句解析部はコメントと予約語を表す正規表現を用いてコメント除去を行う．本ツールの字句解析を，コメント除去，字句分割，識別子判別の三つの処理に細分化した．以降，各処理の詳細について記述する．

3.1.1 コメント除去と文字列リテラルの識別

コードクローンにはコメントや空白は含まれないという定義に基づき，入力されたソースコードのコメントを除去する処理である．コメント除去に使用する手法としては，ソースコード中からコメントを表す正規表現にマッチする文字列を選び出し，同じ文字数の空白に置換している．

文字列リテラル内にコメント記号が書かれた場合，コメントではなく文字列リテラルであると認識する必要がある [15], [16]．図 2 は，C 言語のソースコード

```
1 char *CommentStart = "/*";  /* Comment1 */
2 char *CommentEnd   = "*/"; /* Comment2 */
```

図 2　文字列リテラル内にコメント記号が含まれる例

の一部である．このソースコードに対し，C 言語コメントの文法を表す正規表現(注4)のみを用いてマッチを行った場合，コメントを表す部分に正しくマッチしない．マッチングしたい文字列は青字の部分であるが，実際にマッチした部分の開始地点はダブルクオーテーションに囲まれた /* になってしまう．この対策として，文字列リテラルを正規表現を用いて識別する必要がある．

3.1.2　字句分割

コメント除去が行われたソースコードに対して，字句分割を行う．字句分割で使われるルールは以下のとおりである．番号が小さいルールほど優先される．

（1）　文字，文字列リテラルは 1 字句とする．

（2）　空白，タブ文字と改行の前後で字句を分割する．

（3）　記号は 1 文字ずつで分割する．記号が複数文字で一つの意味を表す場合でも，1 文字で 1 字句とする．

（4）　それ以外の連続した英数字列は 1 字句とする．

3.1.3　識別子判別・変換処理

識別子判別では字句分割で英数字列として分割された字句が，識別子か予約語かを判定するものである．予約語は変数名や関数名に使用できない文字列のことを指す．予約語の集合は，プログラミング言語ごとにそれぞれ定義されている．変換処理は CCFinderX の処理と同様に，識別子を同一の字句に変換するものである．

3.1.4　クローン検出・出力整形

クローン検出では変換された字句列を比較し，コードクローンを検出する．CCFinderSW の開発には，Ngram を用いたアルゴリズムを採用している [17]．最後に出力整形では，検出されたコードクローンをクローンペアとして，出現するファイル・行番号などを出力する．

3.2　構文定義記述解析モジュールの開発

本研究で開発した構文定義記述解析モジュールは，ANTLR の構文定義記述を構文解析し，生成した構文木から必要な情報を取得する．この構文解析には，

(注4)：例えば，/\ * [\s\S]*?\ * /|//((?![\r\n])[\s\S])*

```
1 grammar Prog;
2 prog: expr;
3 expr: term ( ( '+' | \ '-' ) term )*;
4 term: factor ( ( '*' | \ '/'  ) factor )*;
5 factor: INT | \ '(' expr ')';
6 INT: [0-9]+;
7 WS: [\t\r\n ]+ -> \ skip;
```

図 3　四則演算式を表す構文定義記述

Java で動作する構文解析器を ANTLR で生成して組み込んだ．

以降，**3.2.1** では ANTLR の構文定義記述について，及び本研究で利用した理由について説明する．**3.2** では新たに開発した構文定義記述解析モジュールの実装について説明する．**3.2.2** から **3.2.4** ではコメント・文字列リテラル・予約語のそれぞれにおける，正規表現への変換手法について説明する．

3.2.1　構文解析器生成系 ANTLR の構文定義記述の利用

本節では，本研究で利用した ANTLR の構文定義記述の文法と，調査を行った構文定義記述に頻出した表現方法について説明する．

構文解析器生成系は，字句解析器や構文解析器を自動的に生成するプログラムであり，パーサジェネレータとも呼ばれる．構文解析器生成系の一つである ANTLR は，構文木の構築・探索が可能な構文解析器を生成する．Twitter(注5)での検索クエリの解析に採用されるなど，ANTLR は広く使用されているため [18]，本研究の対象として選択した．

ここで，図 3 に ANTLR の構文定義記述の例を示し，用いられる文法について説明する．まず，以下の構文定義記述の 1 行目では，**grammar Prog** と書くことで構文定義記述が表す文法の名前が **Prog** であることを宣言している．2 行目以降は文法を構成するルールについて記述している．先に示した四則演算式を表す記述の中では，字句解析ルールとして ‘INT’ と ‘WS’ が存在する．‘INT’ は数字列を表していて，‘WS’ は空白とタブ文字と改行を表している．

一つのルールは **‘ルール名:ルールブロック’** のように記述され，このようなルールを複数組み合わせることで一つの文法を形成する．ANTLR の文法を定義するルールには，字句解析ルールと構文解析ルールの二つが存在する．字句解析ルールの名前は大文字から

(注5)：https://twitter.com/

表 2　ANTLR で使用される主要な字句

字句	説明
‘リテラル’	文字，または文字列にマッチする．
[文字集合]	指定された文字のどれか一つにマッチする．a-z のように書くことで範囲を指定することも可能．正規表現で使用されるものと同様．
.	任意の一文字にマッチする
∼x	x で記述されている集合にマッチしない任意の一文字にマッチする．x は 1 文字のリテラルや文字集合が指定される．本論文では NOT 演算子と呼ぶ
x*	x の 0 回以上の繰り返しにマッチする．
x?	x の 0 回，または 1 回の出現にマッチする．
x*?	x の 0 回以上の最短の繰り返しにマッチする．
x|y	x または y にマッチする．

始まり，構文解析ルールの名前は小文字から始まる．ANTLR で用いられる表現については，ANTLR の開発者によるドキュメントが存在する(注6)．その中から主要な字句解析ルールを抜粋し，表 2 に示す．

次に，ANTLR で使用されるコマンドについて説明する．ANTLR は字句解析ルールを用いてソースコード中で出現する字句を定義するが，それぞれの字句にコマンドを指定し，特定の操作を施すことができる．

コマンドは，通常のルールの後に‘->’とコマンド名を加えて記述する．本研究で扱ったコマンドは二つである．一つ目は skip である．skip を指定された字句は，字句解析によって読み飛ばされる．二つ目は channel コマンドである．channel(x) のように記述し，x にはあらかじめ定義された channel 名が入る．特に channel(HIDDEN) は skip と同じ処理を行い，このコマンドを指定された字句は ANTLR の Parser によって無視される [19]．

3.2.2　コメント文法の正規表現への変換

本節では ANTLR の構文定義記述から，コメントを表す正規表現の抽出方法について述べる．文法情報を正規表現で抽出した理由としては，一般的に正規表現は文字列の置換処理などに用いられることが多く，提案ツールの実装言語である Java においても，正規表現を用いた文字列処理に関する標準ライブラリが存在するためである．また，ANTLR の構文定義記述は正規表現と近い表現が用いられており，変換が容易であるためである．更に，コメント除去に正規表現を用いることで，既存ツールで採用されている分類以外のコメント文法にも幅広く対応できると考えている．

コメント文法の正規表現への変換は以下の四つの Step で行われる．

```
1  Comment:'/*'.*?'*/';
2  Block1: '/*'.*?'*/' -> skip;
3  Block2: '/*'.*?'*/' -> channel(HIDDEN);
4  Block3: '/*'.*?'*/' -> channel(BComment);
```

図 4　コメントの定義例

Step A　全てのルールの中から，コメントを表すルールを選びだす．

Step B　別のルールを参照している部分を再帰的に適用する．

Step C　Java で使用される正規表現に変換する．

Step D　生成された正規表現を全て結合して一つの表現にする．

各 Step の詳細は以下のとおりである．

Step A　まず，全てのルールの中からコメントを表すルールを選び出す．判断基準を四つ設け，そのうちの少なくとも一つに当てはまることでコメントを表すルールとして識別した．その四つの判断基準を以下に示し，それぞれに当てはまるルールを図 4 に例示する．例示された四つのルールは，いずれも C 言語の複数行コメントに相当する表現である．

（1）　ルール名に ‘comment’ という文字列が含まれている．このとき，大文字と小文字の差異は無視する．

（2）　skip が呼ばれている．

（3）　channel(HIDDEN) が呼ばれている．

（4）　channel が呼ばれていて，channel 名に ‘comment’ という文字列が含まれている．このとき，大文字と小文字の差異は無視する．

Step B　選び出されたルールの中で，別のルールを参照している部分を再帰的に適用する．**3.2.1** で示した四則演算式を表す構文定義記述の例では，term という名前のルールの中で factor というルールが埋め込まれている．このようにルールの中で他のルールが参照されている場合は，Step C で正規表現に変換するために参照先のルールの内容を再帰的に適用する．正規表現で表現できないコメント文法を解析する場合などに，再帰的適用が停止しない場合があるため，ルールを適用するたびに二つ組（ルール，対象の記号）を記録する．この記録を用いて，過去に同一の二つ組で表される適用が行われていれば再帰的適用を停止する．

Step C　Step B で適用されたルールを，CCFinderSW の開発言語である Java で使用可能な正規表現に変換する．正規表現と構文定義記述で用いられる表

(注6)：https://github.com/antlr/antlr4/blob/master/doc

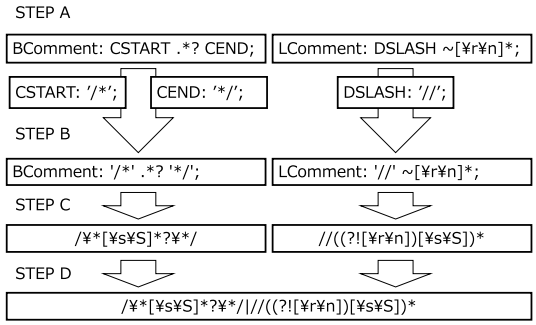

図 5　コメント記述ルールの正規表現への変換例

現の違いで，変換が必要なものは三つある．

一つ目はシングルクォーテーションである．ANTLR の構文定義記述では，対象言語に出現する実際のリテラルをシングルクォーテーションで囲んで記述する．正規表現ではこのシングルクォーテーションは不要であるため除去する．二つ目は NOT 演算子である．ANTLR の構文定義記述では，ある特定の文字集合にマッチしない集合を表すために先頭に，NOT 演算子の役割をする ‘~’ をつける．この NOT 演算子に直接的に相当するものは正規表現には存在しないため，正規表現の否定的先読みを用いて同等の表現に変換する．三つ目は ‘.’ である．ANTLR の構文定義記述では ‘.’ は任意の 1 文字を表すのに対し，Java で使用される正規表現では改行以外の任意の 1 文字を表し，定義が異なっている．この定義の差異を埋めるため，構文定義記述で用いられる ‘.’ を正規表現での同等の表現である ‘[\s\S]’ に変換することで対応している．

Step D　生成された正規表現を全て結合して一つの表現にする．これは Step C で生成された全ての正規表現の間に，論理和を表す記号である ‘|’ を挟んで結合することで行われる．

図 5 は，コメントに相当するルールの正規表現への変換例である．まず Step A では判断基準に基づいて，*BComment* と *LComment* というルールを選択する．次に Step B ではこの選択されたルールで，他のルールを参照している部分を再帰的に適用する．*BComment* では *CSTART* と *CEND* という他のルールへの参照があるため，参照先のルールを代入して他のルール名が含まれない表現にする．Step C では Java で使用可能な正規表現に変換し，Step D で全ての正規表現を結合する．

3.2.3　文字列リテラル文法の正規表現への変換

本節では ANTLR の構文定義記述から文字列リテラル文法を抽出し，正規表現へ変換する方法について述べる．

```
1  StringLiteral: QUOTE StringCharacters? QUOTE;
2  STRING : 'string';
```

図 6　文字列リテラルの定義例

```
1  WHILE:'while';\\
2  WHILE:[wW][hH][iI][lL][eE];
```

図 7　予約語の定義例

本研究では，ANTLR の構文定義記述内での文字列リテラルの記述法を調査した．図 6 に示すのは，調査対象の中に多く存在したものである．

1 行目のルールでは，ルール名に ‘String’ という文字列が含まれており，文字列リテラルが定義されている．2 行目のルールは，ルール名に ‘STRING’ が含まれているが予約語の定義である．1 行目のようなルールだけを抽出するために，文字列リテラルのルールの判断基準を,「ルール名に ‘STRING’ という文字が含まれるもののうち，予約語の定義ではないルール」とした．このとき，大文字と小文字の差異は無視する．

構文定義記述から正規表現への変換は，コメントの場合と同様であるため説明を省略する．

3.2.4　予約語一覧の正規表現への変換

本節では ANTLR の構文定義記述から予約語一覧を抽出し，正規表現へ変換する方法について述べる．

ANTLR の構文定義記述内での予約語の記述法を調査した結果，大きく分けて 2 種類の記述法があった．以下の図 7 に，*while* という予約語の定義に対する 2 種類の記述法を例示する．

1 行目のルールでは，*WHILE* というルール名に *while* という単語が紐付けられている．この記述法は最も広く使われているものであった．そして二つ目のルールでは文字クラスを用いた記述法が使用されている．これは *while* に含まれる 5 文字のそれぞれに大文字と小文字のどちらでもマッチすることを許す記述法である．このような 2 種類の記述法を抽出するために，予約語一覧の正規表現への変換は以下の五つの Step で行われる．

Step α　全てのルールで，出現しているリテラルのうち英字列に一致するものを抽出する．

Step β　全てのルールで，別のルールを参照している部分に再帰的に適用する．

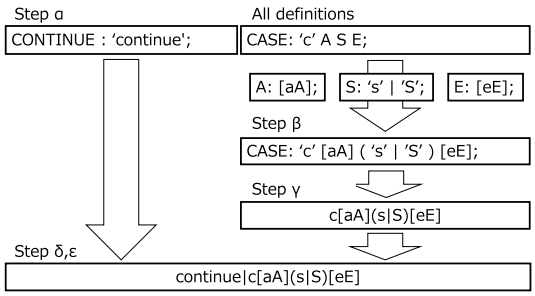

図 8　予約語一覧の正規表現への変換例

Step γ　Step β で適用されたルールを Java で使用可能な正規表現に変換する．

Step δ　Step γ で変換された正規表現で，英字列を表しているものを選出する．

Step ϵ　Step α で抽出されたリテラルと Step δ で選出された正規表現を全て結合して一つの表現にする．

図 8 は，予約語一覧の正規表現への変換例である．**3. 2. 2** で説明したコメントの場合と同様であるため，説明は省略する．

4. 評 価 実 験

本章では CCFinderSW を評価するために行った三つの実験について説明する．

4. 1　構文定義記述解析モジュールを用いた文法情報抽出実験

本研究で開発したモジュールを用いて，どの構文定義記述からコメントと文字列リテラルと予約語の情報が抽出可能であるかを確認するための実験を行った．

実験の対象となるファイルは GitHub のリポジトリである grammars-v4(注7) を使用した．このリポジトリは ANTLR の構文定義記述を集めたものであり，約 150 種類が含まれている．ANTLR の開発者である Parr も含めて，170 人以上の貢献者が存在し，現在でも更新が続けられている．実験で使用したリポジトリのスナップショットは 2017 年 12 月 14 日時点のものである．

本研究では，リポジトリに含まれている 154 種類の文法の中から，GitHub の Advanced Search(注8) の検索対象に登録されている言語に対応する，構文定義記述を実験対象とした．次に 42 の構文定義記述からコメント，文字列リテラル，予約語の三つの文法情報をあらかじめ手作業で記録し，構文定義記述解析モジュールが正しく情報を抽出できるかどうかを判定した．

本研究における予約語の定義について記述する．予約語とは，変数名や関数名に使用できない文字列のことを指し，言語によって定められている．一方，プログラミング言語におけるキーワードとは，特別な役割をもつ文字列のことを指す．予約語とキーワードは似通った存在と言われているが，言語によってはキーワードであっても予約語ではない文字列が存在し，また予約語が存在しない言語も存在する．この実験では，各言語のキーワードは予約語であると定義して実験を行った．

表 3 は，構文定義記述のファイル名と三つの情報を抽出可否を示したものである．○は抽出可能，×は抽出不可能を示し，「-」は構文定義記述に対応する記述がなかったことを示している．表 3 より，本研究で開発したモジュールではプログラミング言語の文法を表す 42 の構文定義記述のうち，コメントは 93%，予約語は 98%，文字列は 88%から抽出することができた．もともと文法情報が定義されていないものも含めて，三つとも抽出できたものは 34 言語で，これは全体の 81%にあたる．

本実験で抽出不可能となった構文定義の例について，図 9 を用いて説明する．図 9 の左側は Lua.g4 におけるコメントを定義する字句解析ルールを表し，右側は生成されるコメントを例示したものである．この例のとおり，Lua.g4 のコメントは，開始時と終了時の '=' の数が同一であれば '=' の数を変化させても成立する．一般的な正規表現は，文字の繰り返し数を記憶し，その後に出現する文字列の照合に用いることはできない [15]．そのため，開始時の '=' の数を記憶し，終了時の '=' の数が同一であるかを判定することが正規表現はできないため，抽出不可能とした．なお，CCFinderSW は **3. 2. 2** の Step B において，図 9 の `COMMENT` に対して `NESTED_STR: '[' .*? ']'` のみを適用した記号列，及び `NESTED_STR: '=' NESTED_STR '='` と `NESTED_STR: '[' .*? ']'` の両者を適用した記号列を生成し，ルールの適用を終了する．そのため，Step C において '=' の数が 0〜1 個の場合にのみ抽出できる正規表現が生成されるが，2 個以上のときは抽出できないため，本実験では抽出不可能とした．

4. 2　C++におけるコードクローン検出結果の CCFinderX との比較実験

本節では，C++で記述されたソースコードに対す

(注7)：https://github.com/antlr/grammars-v4
(注8)：https://github.com/search/advanced

表 3　文法情報抽出実験の対象言語と実験結果

ファイル名	コメント	予約語	文字列	ファイル名	コメント	予約語	文字列	ファイル名	コメント	予約語	文字列
agc	○	○	-	fortran77	○	○	-	protobuf3	○	○	×
antlrv4	○	○	○	golang	○	○	○	python3	○	○	○
apex	○	○	○	html	○	-	○	r	○	-	○
asm6502	○	○	○	idl	○	○	○	rexx	×	○	○
aspectj	○	○	○	java9	○	○	○	scala	○	○	○
c	○	○	○	kotlin	○	○	×	smalltalk	○	○	○
clojure	○	○	○	lua	×	○	○	smtlibv2	○	○	○
cobol85	○	○	○	modelica	○	○	○	swift3	○	○	○
cool	○	○	○	m2pim4	○	○	○	vba	○	○	○
cpp14	○	○	○	objective-c	○	○	○	verilog2001	○	○	○
csharp	○	○	×	pascal	○	○	○	vhdl	○	○	○
css3	○	×	○	phplexer	×	○	×	visualbasic6	○	○	○
ecmascript	○	○	○	plsql	○	○	○	webidl	○	○	○
erlang	○	○	○	prolog	-	-	×	xml	○	-	○

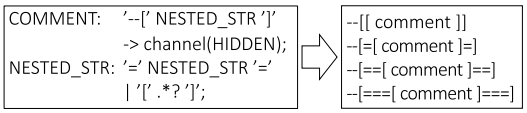

図 9　抽出不可能なコメントのルール

るコードクローン検出において，既存のコードクローン検出ツールである CCFinderX との検出結果の比較と分析を行う．この実験の目的は，CCFinderX が対応しているプログラミング言語において，CCFinderSW が同様のコードクローン検出を行うことを示すことにある．既存のコードクローン検出ツールの中から CCFinderX を比較対象として選択した理由は，CCFinderSW と同様に字句単位の検出方法を採用しており，広く使用されている代表的なツールであるからである [20], [21].

まず，実験の手順について説明する．同一のソースコードに対して，CCFinderX と CCFinderSW でコードクローン検出を行う．この際，二つのツールで用いられるパラメータや条件は可能な限り揃える．次に，それぞれのツールで検出されたクローンペアを比較し，同一または類似したクローンペアが検出されているかどうかを確認する．本研究では，二つのツールで検出されたクローンペア間の類似度が，しきい値以上であれば一致しているとみなす．特に同一であるとみなされたものは完全一致と表現する．一致するものが存在しない場合，つまり片方のツールでしかクローンペアが検出されていない場合は，もう片方のツールで検出されない原因を分析する．また，それぞれで検出されたクローンペア数と，マッチしたクローンペア数について測定し，その割合を算出する．

本実験の対象言語は，CCFinderX の対応言語の一つである C++を選択した．実験対象となるソースコードとして，git(注9) の master ブランチから 2018 年 12 月 21 日 0 時 0 分時点でのスナップショットを取得した．

次に，CCFinderX と CCFinderSW に与えるオプションについて説明する．共通のオプションとして検出クローン片の最低字句数のしきい値を，大規模ソースコードを対象とした場合に使用される 100（およそ 15〜20 行程度）にした [22]．CCFinderX では実用的なクローンを出力するために，メトリクスなどを用いた様々なフィルタリングや，構文的な解析に基づく正規化を行っている．このため CCFinderX は，多言語に適用可能な字句分割と検出手法を用いる CCFinderSW よりも少ない量のクローンペアを出力する．本実験では，可能な限り CCFinderX と CCFinderSW との検出条件を揃えるために，CCFinderX に与える三つのオプションを，デフォルトの値から変更した．まず P-match フィルタ(注10) [23] をデフォルトでオンのところをオフにした．また，クローン片に含まれる最低字句種類数のしきい値をデフォルト値の 12 から 0 に変更し，Block Shaper(注11) [24] に関するオプションをデフォルト値の 2 からを 0 に変更した．また，正確な比較を行うために，CCFinderSW を実行する前に CCFinderX が無視する初期化テーブルを対象ソースコードから除去した．CCFinderX が初期化テーブ

(注9)：https://github.com/git/git
(注10)：クローンペア間で変数名が対応関係をもたない場合，そのクローンペアを除去するフィルタ．詳細は [23] 参照．
(注11)：ブロック単位のコードクローン以外を除去するフィルタ [24] を指す．Block Shaper オプションに 2 を設定するとブロック単位のコードクローン以外を全て除去する動作になり，0 を設定するとフィルタが無効になり除去を行わない動作になる．

ルを特定し検出対象か取り除く厳密なアルゴリズムについて，説明した文献やドキュメントが存在しないため，この前処理は著者が独自に実装した．

次に，検出されたクローンペアの比較方法の詳細と一致の定義について説明する．あるファイル A とファイル B 間に存在するクローンペアについて，CCFinderSW と CCFinderX で検出されたものを比較を行うとする．まずクローン片のそれぞれの名前についての定義として，CCFinderSW で検出されたクローンペアのクローン片を *SWA* と *SWB* として，CCFinderX で検出された *XA* と *XB* とする．本研究で用いる手法は行単位の一致率の計算を行う．比較するときに用いる数値としては，*SWA* と *XA* のそれぞれのクローン片の開始行と終了行，そして *SWB* と *XB* のそれぞれのクローン片の開始行と終了行となる．ここでそれぞれのクローン片の開始行と終了行の値を *start* と *end* を名前の後につけることで表現し，クローン片の長さを *len* をつけることで表現する．以上のことから，それぞれのクローン片の長さは以下のような式となる．

$$SWAlen = SWAend - SWAstart + 1 \quad (1)$$

$$SWBlen = SWBend - SWBstart + 1 \quad (2)$$

$$XAlen = XAend - XAstart + 1 \quad (3)$$

$$XBlen = XBend - XBstart + 1 \quad (4)$$

次に，このクローン片の一致率の計算式を説明する．*SWA* と *XA* で一致している部を *MA* と表現し，ファイル B においても同様とする．このとき，*MA* の長さの定義を，与えられた二つの値で大きい側の値の返す関数 Max と，与えられた二つの値で小さい側の値を返す Min を用いて，以下のように定義できる．*MB* に対しても同様の計算をする．

$$MAlen = Min(SWAend, XAend) - Max(SWAstart, XAstart) + 1 \quad (5)$$

$$MBlen = Min(SWBend, XBend) - Max(SWBstart, XBstart) + 1 \quad (6)$$

次に *SWA* と *XA* の一致率を表す $MatchA(\%)$ を計算し，ファイル B においても $MatchB(\%)$ を計算する．最後に CCFinderSW と CCFinderX で検出されたクローンペアの一致率として，$MatchSWX(\%)$ を計算する．

$$MatchA = \frac{MAlen * 100}{Max(SWAlen, XAlen)} \quad (7)$$

$$MatchB = \frac{MBlen * 100}{Max(SWBlen, XBlen)} \quad (8)$$

$$MatchSWX = Min(MatchA, MatchB) \quad (9)$$

表 4 は実験の結果を示したものである．この表は，CCFinderX と CCFinderSW の検出クローンペア数，その二つのツールで検出されたクローンペアの一致数を表したものである．二つのツール間で完全一致したクローンペア数は 1806 個である．検出ペア中の一致した割合は，一致した数を検出ペアの数で割った値である．検出ペア中の一致した割合の列を見ると，CCFinderX で検出されているクローンペアの約 98%が CCFinderSW で検出されており，逆も同様に約 98%であることから，CCFinderSW は CCFinderX とほぼ同等の検出能力をもっているといえる．

次に，CCFinderX と CCFinderSW のそれぞれで一致が見られなかったクローン片を目視で確認し，分析した結果について述べる．まず，CCFinderX のみで検出された 32 個のクローンペアについて，CCFinderSW で検出されなかった五つの原因に説明する．各項目末尾に記載した括弧内の数値は，該当したクローンペア数を表す．

（1） 何らかの理由で CCFinderX で検出されるクローンペアが極大クローンペア（**2.** 参照）ではないため（16 個）．

（2） CCFinderX で検出されたコードクローンが，CCFinderSW での検出では字句数がしきい値の 100 より下回っていたため（8 個）．

（3） CCFinderX が，unsigned を含む型定義を一つの字句として認識しているため（4 個）．

（4） CCFinderX は，二項演算子 '+' によりオペランドの文字列の連結を行っている場合に一つの字句として認識する機能があるが，CCFinderSW ではその機能がないため（2 個）．

（5） CCFinderSW で検出対象のソースコードに行った前処理が誤っていたため（2 個）．

（2）の現象が生じた理由は，CCFinderSW と CCFinderX 間で字句数の算出方法が異なるからである．CCFinderX は各言語ごとに手作業で字句解析器を実装している．そのため，言語のよらず **3. 1. 2** のルールに基づき字句数の算出を行う CCFinderSW は，CCFinderX とは字句数の算出結果が異なることがある．また，全ての場合において 1 字句不足しているため，検出できなかった．（3）の前処理は，CCFinderSW

表 4　CCFinderX との出力クローンペアの比較結果

	検出ペア	一致したペア	一致した割合	一致なし
CCFinderX	1928	1896	0.983	32
CCFinderSW	1944	1904	0.979	40

を実行する前に CCFinderX が無視する初期化テーブルを対象ソースコードから除去するために実施した．しかし，CCFinderX よりも多くのコード片を初期化テーブルとして特定し，除去したことがあったため，検出結果に差異が生じた．この原因は，CCFinderX が初期化テーブルを特定し検出対象か取り除く厳密なアルゴリズムについて，説明した文献やドキュメントが存在しないため，やむを得ずこの前処理を著者が独自に実装したことにある．この独自に実装した前処理において，CCFinderX よりも多くのコード片を初期化テーブルとして特定し，除去した場合があった．

次に CCFinderSW のみで検出された 40 個のクローンペアについて，CCFinderX で検出されなかった三つの原因に説明する．各項目末尾に記載した括弧内の数値は，該当したクローンペア数を表す．

（1） 何らかの理由で CCFinderX で検出されるクローンペアが極大クローンペア（**2.** 参照）ではないため（16 個）．

（2） CCFinderSW で検出されたコードクローンが，CCFinderX での検出では字句数がしきい値の 100 より下回っていたため（22 個）．

（3） CCFinderSW で検出対象のソースコードに行った前処理が誤っていたため（2 個）．

（2）の現象が生じた理由は，CCFinderSW で検出できなかった原因と同様に，CCFinderSW と CCFinderX 間で字句数の算出方法が異なることにある，また，（3）の誤りについても，CCFinderSW で検出できなかった原因と同様に，やむを得ず著者が前処理を独自に実装したことにある．

以上の実験と分析から，CCFinderX と CCFinderSW の検出能力はほぼ同等であると考えられる．検出結果に差異があったものに関しては目視で原因を突き止めることができた．

4.3 Verilog HDL に対するコードクローン検出精度に関する実験

本節では，Verilog HDL で記述されたソースコードに対して開発した CCFinderSW を用いてコードクローン検出を行い，その検出精度を評価する実験について説明する．この実験は，CCFinderX などのコードクローン検出ツールが一般的に対応していない言語に対しても，CCFinderSW を用いることでタイプ 2 のコードクローンが検出可能であることを示すために行った．

上村らは，代表的な HDL である Verilog HDL のコードクローンの検出手法を提案し，10 件のプロジェクト中のコードクローンについて調査している [25]．この提案する検出手法は，Verilog HDL のソースコードを幾つかの変換規則に基づいて疑似 C++に変換し，CCFinderX で変換後のソースコードを C++のソースコードとしてコードクローン検出を行うものである．

本研究における，提案ツールである CCFinderSW の検出手法の評価に，上村らが用いたコードクローン検出ツールの評価手法を用いる [26]．コードクローンの検出精度は，検出されたコードクローンのうち正解の割合（Precision）と，全ての正解のうち検出できたコードクローンの割合（Recall）で評価することができる．Svajenko らは，自明なコードクローンを埋め込むことで正解集合を構築する手法を提案している [27]．この手法では，まずソースコード中から関数やコードブロック単位でコード片を複数選択し，これに対して複数の変異を適用しソースコード中に埋め込んでいる．そして，この変異コード片を文法上問題のない，ランダムな位置に挿入し，自明なコードクローンを生成し，正解コードクローンとして記録する．最後に，評価対象のツールが，正解コードクローンを正しく検出できた割合を測定する．上村らは，Svajenko らが用いた変異手法に幾つかの変更を加えて，Verilog HDL のソースコードに対するコードクローン検出の提案手法を評価している．本実験では，上村らが用いたデータセットと同じものを用いた

Svajlenko と Roy は，抽出するコード片の粒度を関数及びブロック単位とすると述べている．これに基づき上村らは **module**，**always**，**if**，**case** の 4 種類のブロックを対象にしている．この評価対象には，Verilog HDL のプロジェクトの ridecore(注12) が選ばれている．

本研究では，上村らが作成した変異コードが埋め込まれたソースコードに対して，CCFinderSW を用い

(注12)：https://github.com/ridecore/ridecore.git

表 5　Precision と Recall の測定結果

	Precision					Recall				
(%)	module	always	if	case	合計	module	always	if	case	合計
タイプ 1	100	100	100	100	100	99	99	99	99	99
タイプ 2	100	100	100	100	100	98	100	100	100	99
総計	100	100	100	100	100	99	99	99	99	99

てコードクローン検出を行い，Precision と Recall を測定する．この際，CCFinderSW に与える構文定義記述は，grammars-v4 の Verilog2001.g4 を使用して検出を行った．Precision は，CCFinderSW によって検出されたコードクローンのうち，誤検出ではないと判定された割合を表している．第一著者が手作業で検出されたコードクローンを確認し誤検出かどうか判定した．このコードクローン検出では，CCFinderSW の最低字句数をデフォルト値である 50 に設定したときに検出されたコードクローンを，クローン片に含まれる字句種類数の最低しきい値を 12 としてフィルタリングを行ったものを対象とした．Recall は，変異コード片を用いて埋め込まれた正解コードクローンのうち検出されたものの割合を表している．上村らは，**module**，**always** ブロックのコード片に対しては最低字句数を 50（およそ 10 行程度），**if**，**case** ブロックのコード片に対しては最低字句数を 25 としており，本実験でもこの値を採用した．

本ツールで検出した結果の Precision と Recall を表 5 に示す．総計はタイプ 1 とタイプ 2 を区別なく集計した結果である．本ツールの検出結果では，全体の Precision は 100%，Recall は 99%となり，Precision が 99%，Recall が 93%の上村らの手法 [25] より高い数値となった．Recall が 100%にならない理由は，CCFinderSW は識別子名は記号を含まないという前提で **3. 1. 2** で述べた字句分割を行うが，Verilog HDL の変数名はグレーブアクセント（’）を含むことができるからである．対象とした Verilog HDL のソースコード中に，グレーブアクセントを含むため適切に正規化できなかった変数名が一つ存在した．識別子名に記号が含むまれる前提で字句分割を行う方式に変更することで解決できる可能性があるが，他言語のソースコードに適用した際に正確な検出ができなくなる可能性があるため，どちらの方式が有効であるか比較実験を通して評価を行う必要がある．

5. 考　　察

本ツールが構文定義記述に基づいて行うコメント除去方式と識別子変換方式については，コメント除去及び識別子変換を行う他のコードクローン検出ツールにおいても利用できる可能性があると考えられる．ただし，コードクローン検出ツールごとにトークン列やコードクローンの内部表現が異なることが多いため，CCFinderSW のコメント除去及び識別子変換の実装を再利用することは容易ではないと考えられる．そのため，他のコードクローン検出において，CCFinderSW のコメント除去変換及び識別子変換方式を実装する場合は，拡張元のトークン列やコードクローンの内部表現にあわせた実装を行う必要がある．

6. む　す　び

本研究では字句単位のコードクローン検出における字句解析に必要な文法情報を，構文解析器生成系の構文定義ファイルから自動的に抽出するモジュールを開発した．そしてこのモジュールを用いて，ANTLR の構文定義記述を入力として与えることで，対象言語の文法に沿ったコードクローン検出が可能な CCFinderSW を開発した．

構文定義記述解析モジュールの問題点として，構文定義記述は対象となるプログラミング言語の文法に依存するが，書き手にも依存する．つまり，同じ文法であっても複数の記述法で表現することができる．本研究での開発は，構文定義記述の調査において多く存在した記述法に対して行われたものである．新たに異なる記述法に対応するための，モジュールの拡張が必要である．

謝辞　本研究は JSPS 科研費 JP19K20240, JP18H04094 の助成を受けた．

文　　献

[1] 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，vol.18, no.5, pp.47–54, 2001.

[2] 肥後芳樹，神谷年洋，楠本真二，井上克郎，“コードクローンを対象としたリファクタリング支援環境,” 信学論（D-I），vol.J88-D-I, no.2, pp.186–195, Feb. 2005.

[3] 肥後芳樹，楠本真二，井上克郎，“コードクローン検出とその関連技術,” 信学論（D），vol.J91-D, no.6, pp.1465–

1481, June 2008.
[4] Y. Semura, N. Yoshida, E. Choi, and K. Inoue, “Multilingual detection of code clones using antlr grammar definitions,” Proc. APSEC 2018, pp.673–677, 2018.
[5] T. Kamiya, S. Kusumoto, and K. Inoue, “CCFinder: A multilinguistic token-based code clone detection system for large scale source code,” IEEE Trans. Softw. Eng., vol.28, no.7, pp.654–670, 2002.
[6] 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,” 信学技報，SS2001-14, 2001.
[7] Y. Yamanaka, E. Choi, N. Yoshida, K. Inoue, and T. Sano, “Industrial application of clone change management system,” Proc. IWSC 2012, pp.67–71, 2012.
[8] K. Inoue, Y. Higo, N. Yoshida, E. Choi, S. Kusumoto, K. Kim, W. Park, and E. Lee, “Experience of finding inconsistently-changed bugs in code clones of mobile software,” Proc. IWSC 2012, pp.94–95, 2012.
[9] K. Sakamoto, K. Shimojo, R. Takasawa, H. Washizaki, and Y. Fukazawa, “OCCF: A framework for developing test coverage measurement tools supporting multiple programming languages,” Proc. ICST 2013, pp.422–430, 2013.
[10] C.K. Roy, J.R. Cordy, and R. Koschke, “Comparison and evaluation of code clone detection techniques and tools: A qualitative approach,” Science of Computer Programming, vol.74, no.7, pp.470–495, 2009.
[11] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, “DECKARD: Scalable and accurate tree-based detection of code clones,” Proc. ICSE 2007, pp.96–105, 2007.
[12] H. Sajnani, V. Saini, J. Svajlenko, C.K. Roy, and C.V. Lopes, “SourcererCC: Scaling code clone detection to big-code,” Proc. ICSE 2016, pp.1157–1168, 2016.
[13] V. Saini, F. Farmahinifarahani, Y. Lu, P. Baldi, and C.V. Lopes, “Oreo: Detection of clones in the twilight zone,” Proc. ESEC/FSE 2018, pp.354–365, 2018.
[14] G. Zhao and J. Huang, “DeepSim: Deep learning code functional similarity,” Proc. ESEC/FSE 2018, pp.141–151, 2018.
[15] J.E. Friedl, Mastering regular expressions, O’Reilly Media, 2002.
[16] 瀬村雄一，多様なプログラミング言語に対応可能なコードクローン検出ツール CCFinderSW，大阪大学大学院情報科学研究科 修士学位論文，（オンライン），入手先 (http://sel.ist.osaka-u.ac.jp/lab-db/Mthesis/contents.ja/146.html)，2019.
[17] Y. Semura, N. Yoshida, E. Choi, and K. Inoue, “CCFinderSW: Clone detection tool with flexible multilingual tokenization,” Proc. APSEC 2017, pp.654–659, 2017.
[18] T. Parr, “About The ANTLR Parser Generator,” http://www.antlr.org/about.html.
[19] T. Parr, The definitive ANTLR 4 reference, The Pragmatic Bookshelf, 2013.
[20] D. Rattan, R. Bhatia, and M. Singh, “Software clone detection: A systematic review,” Information and Software Technology, vol.55, no.7, pp.1165–1199, 2013.
[21] C.K. Roy and J.R. Cordy, “A survey on software clone detection research,” Technical report, School of Computing, Queen’s University, 2007.
[22] 丸山勝久，沢田篤史，小林隆志，大森隆行，林　晋平，飯田元，吉田則裕，角田雅照，岩政幹人，今井健男，遠藤侑介，村田由香里，位野木万里，白石　崇，長岡武志，林　千博，吉村健太郎，大島敬志，三部良太，福地　豊，“産学連携によるソフトウェア進化パターン収集の試み,” 情処学研報，vol.2014-SE-184/2014-EMB-33, no.1, pp.1–8, May 2014.
[23] B.S. Baker, “On finding duplication and near-duplication in large software systems,” Proc. WCRE 1995, pp.86–95, 1995.
[24] 肥後芳樹，植田泰士，神谷年洋，楠本真二，井上克郎，“コードクローン解析に基づくリファクタリングの試み,” 情処学論，vol.45, no.5, pp.1357–1366, 2004.
[25] K. Uemura, A. Mori, K. Fujiwara, E. Choi, and H. Iida, “Detecting and analyzing code clones in HDL,” Proc. IWSC 2017, pp.1–7, 2017.
[26] 上村恭平，森　彰，藤原賢二，崔　恩瀞，飯田　元，“ハードウェア記述言語におけるコードクローンの定量的調査,” 情処学論，vol.59, no.4, pp.1225–1239, 2018.
[27] J. Svajlenko and C.K. Roy, “Evaluating modern clone detection tools,” Proc. ICSME 2014, pp.321–330, 2014.

（2019 年 6 月 7 日受付，10 月 2 日再受付，
12 月 5 日早期公開）

瀬村　雄一

平成 28 年度大阪大学基礎工学部情報科学科卒業，平成 30 年度大阪大学大阪大学院情報科学研究科博士前期課程修了．現在，日鉄ソリューションズ株式会社に勤務．コードクローン検出に関する研究に従事．

吉田　則裕　（正員）

平成 21 年大阪大学大学院情報科学研究科博士後期課程修了．同年日本学術振興会特別研究員（PD）．平成 22 年奈良先端科学技術大学院大学情報科学研究科助教．平成 26 年名古屋大学大学院情報科学研究科附属組込みシステム研究センター准教授．平成 29 年より同大学大学院情報学研究科附属組込みシステム研究センター准教授（改組による）．博士（情報科学）．コードクローン分析手法やリファクタリング支援手法に関する研究に従事．

崔　　恩瀞　（正員）

平成 27 年大阪大学大学院情報科学研究科博士後期課程修了．同年同大学大学院国際公共政策研究科助教．平成 28 年奈良先端科学技術大学院大学情報科学研究科助教．平成 30 年より同大学先端科学技術研究科助教（改組による）．平成 30 年より京都工芸繊維大学情報工学・人間科学系助教．博士（情報科学）．コードクローン管理やリファクタリング支援手法に関する研究に従事．

井上　克郎　（正員：フェロー）

昭和 59 年大阪大学大学院基礎工学研究科博士後期課程修了（工学博士）．同年大阪大学基礎工学部情報工学科助手．昭和 59 年～61 年，ハワイ大学マノア校コンピュータサイエンス学科助教授．平成 3 年大阪大学基礎工学部助教授．平成 7 年同学部教授．平成 14 年より大阪大学大学院情報科学研究科教授．ソフトウェア工学，特にコードクローンやコード検索などのプログラム分析や再利用技術の研究に従事．