2017 24th Asia-Pacific Software Engineering Conference

CCFinderSW: Clone Detection Tool with Flexible Multilingual Tokenization

Yuichi Semura*, Norihiro Yoshida†, Eunjong Choi‡ and Katsuro Inoue*
*Osaka University, Japan
{y-semura, inoue}@ist.osaka-u.ac.jp
†Nagoya University, Japan
yoshida@ertl.jp
‡Nara Institute of Science and Technology, Japan
choi@is.naist.jp

***Abstract*—So far, many tools have been developed for the detection of code clones in source code. The existing clone detection tools support only a limited number of programming languages and do not provide any easy extension mechanism to handle additional language. However, from our experience in industry/university collaboration, we found that many practitioners need to analyze source code written in various languages. In this paper, we propose a clone detection tool CCFinderSW that has extension mechanism to handle addition language on demand from practitioners.**

I. Introduction

Programmers often copy and paste code so that they can reuse existing code fragments. This causes code clones, code fragments that are identical or similar code fragments to each other. Generally, a code clone is regarded as one of the factors that hinder software maintainability. For instance, when a cloned code fragment contains a bug, a programmer should check all of its cloned fragments for the same bug. If the cloned fragments contain the same bugs, they should be modified for the same bug. To that end, he/she should know locations of all code clones in the source code. However, it is difficult to for developers to recognize all code clone from large-scale software systems. To alleviate this problem, a multitude of code clone detection tools have been developed. For example, Kamiya has developed a token-based code detection tool CCFinderX [2] that is widely used in academic research as well as industries.

The existing clone detection tools support only a limited number of programming languages and do not provide any easy extension mechanism to handle additional language. From our experience in industry/university collaboration, we found that many practitioners need to analyze source code written in various languages.

In this paper, we propose a clone detection tool CCFinderSW that has extension mechanism to handle addition language on demand from practitioners. It enables the user to easily change the lexical mechanism for comment elimination and identifier replacement.

In the experiment, we compared source code before and after comments elimination of multiple program languages to confirm the flexibility and changeable of lexical analysis mechanism in the CCFinderSW. As a result, it was found that CCFinderSW can remove comments from about 92% of program languages. Moreover, it was confirmed that CCFinderSW is able to detect code clones from all program languages.

II. Overview of CCFinderX

To introduce the mechanism of a token-based clone detection tool, we briefly explain CCFinderX. It identifies not only Type-1 clones (i.e. identical code fragments except for variations in whitespace, layout and comments) but also Type-2 clones (i.e. syntactically identical fragments except for variations in identifiers, literals, types, whitespace, layout and comments.) by replacing identifiers related to types, variables, and constants with a special token.

The code clone detection process of CCFinderX is comprised of the following four steps:

- Step1: (Lexical analysis) Input source code is divided into tokens according to a lexical rule of the programming language. At this step, the white spaces (including comments) between tokens are removed from the source code.
- Step2: (Transformation) The token sequence is transformed based on the transformation rules and then each identifier related to types, variables, and constants is replaced with a special token. This replacement makes code fragments with different variable names to become clone pairs.
- Step3: (Detection) From all the substrings on the transformed token sequence, equivalent pairs are detected as clone pairs using suffix-tree matching algorithm.
- Step4: (Formatting) Each location of clone pair is converted into line numbers on the original source files.

III. A Clone Detection Tool with Flexible Multilingual Tokenization

This section presents an overview of CCFinderSW, a new code clone detection tool using a flexible tokenization mechanism for multilingual program languages.

Figure 1 depicts a clone detection process of CCFinderSW. As you can see in Figure 1, CCFinderSW adopts the same

978-1-5386-3681-7/17 $31.00 © 2017 IEEE
DOI 10.1109/APSEC.2017.80

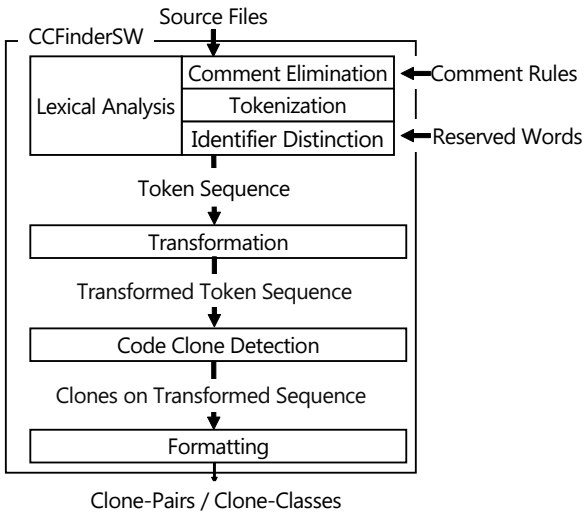

Fig. 1. An Overview of CCFinderSW

four clone detection process, same as the CCFinderX. Furthermore, it also identifies Type-1 and Type-2 code clones.

CCFinderSW provides options for changing target comments for the elimination and reserved words. The option for the comment elimination is used in the lexical analysis process. The option for the reserved words is adopted for finding identifiers.

The reminding subsection details the process of CCFinderSW. The details of the lexical analysis and transformation process of the CCFinderSW are described in Sections III-A and III-B, respectively. In particular, Section III-A explains comment elimination and tokenization mechanisms in order to handle multiple program languages. Next, Section III-C describes the code clone detection process and, finally, Section III-D describes transforming process.

A. Step 1: Lexical analysis

Lexical Analysis process is comprised of three steps namely comment elimination, tokenization, and identifier distinction.

Comments in a programming language are used to annotate/explain the source code. Therefore, during the comment elimination step, they are ignored by default.

In this study, we categorize comments into five comments styles based on their appearance in the source code in different program languages. Hereafter, the details of five comment styles are described with the examples. Note that the red font in each example represents a comment that is ignored during the lexical analysis.

a) End of line comment: End of line comment starts with a symbol (e.g., // in C and Java program) and continues until the end of the line. The following code fragment demonstrates an example of the end of line comment in C and Java programs:

Example of end of line comment

```
v=v+i; //comment
```

b) Multi-line comment: Multi-line comment comments out the source code from a start symbol to an end symbol. The following code fragment illustrates an example of the multi-line comment in C and Java programs, wheter a start symbol is /*, and an end symbol is */ .

Example of multi-line comments

```
v=v+i; /*comment
printf("Hello world"); comment */
printf("Hello world"); /*comment*/
```

In several languages, multi-line comments are nested inside other multi-line comments. The nesting of comment is to describe more multi-line comment in the multi-line comment. The following code fragment describes an example of the nested multi-line comment by other multi-line comments.

Example of nested multi-line comments

```
1 /*
2 /*
3 comment
4 */
5 comment
6 */
```

If nesting is not permitted, the above example can be interpreted as (1) the comment starts with a /* symbol in the 1st line and continues until a */ symbol on the 4th line or (2) the comment starts with a /* symbol in the 1st line and continues until a */ symbol on the 6th line and in both cases, a /* in the second line is ignored. Meanwhile, if nesting is permitted, the above example can be interpreted as a multi-line comment starts with the symbol on the 2nd line and ends at the 4th line, and other comment starts with the symbol on the 1st line and ends on the 6th line. It can be seen from this example that there are big differences in interpreting languages depends on whether nesting is permitted.

c) Full line comment: Full line comment begins with a start symbol and comments out full lines of code. The following code fragment describes an example of the full line comment in Fortran program, where the start symbol is C or * Note that different from the end of line comment, the start symbol does not be interpreted as an indicator for the full line comment, if it appears in the middle of the line.

Example of full line comment

```
c       This is a comment
*       This is a comment
```

d) Full multi-line comment: We defined a comment rule that regards lines from the line where the start symbol exists first to the line where the end symbol exists first as a comment. We named it overall multi-line comment. In Ruby source codes, lines from =begin to lines starting with =end are

TABLE I
SYNTAX INFORMATION FOR EACH COMMENT STYLE

Comment Style	Start symbol	End symbol	Nesting
End of line comment	✓		
Multi-line coment	✓	✓	✓
Full line comment	✓		
Full multi-line Comment	✓	✓	
Character and string literals	✓	✓	

regarded as comments.

Overall multi-line comment

```
=begin comment
puts "Comment 1" comment
puts "Comment 2" comment
=end comment
puts "Hello world"
```

e) Character and String literals: A character literal and string literal is a type of literal in programming for the representation of a value single character and string object within the source code, respectively. For instance, in Java program, characters enclosed in single quotes are identified as a character literal, and characters enclosed in double quotes are identified as a string literal. These literal are not concisely comment styles, they are necessary to distinguish literals from comments. For instance, Therefore, if a character used as a symbol in the comment style appears in the string literal, it is not recognized as the start or end point of the comment. The following code fragment shows an example of a character literal and string literal in Java program. In this example, the characters enclosed in double quotes is the start symbol of a multi-line comment, but they are not regarded as an indicator for the multi-line comment.

Example of Character and String literals

```
String x = " Line Comment start = /* ";
String y = " Line Comment end = */ ";
```

Table I summarizes syntax information of each comment style based on above-mentioned definitions of comment styles. In this table, necessary information is marked with ✓.

After the comments are eliminated from source code based on the defined comment styles, each line of source code is divided into tokens based on a lexical rule. The following four lexical rules are used for the tokenization. Note that a rule with a low number has a higher priority.

1) Each character literal or string literal corresponds to one token, respectively.
2) White spaces and line breaks are delimiters.
3) Each symbol is one token.
4) Other consecutive alphabetic or numeric strings are identified as one token.

B. Step 2: Transformation

In the transformation process, the token sequence is transformed in order to detect meaningful code clones. In detail, all the identifiers representing variable names and function names are replaced by the same token. Reserved words are strings that are reserved by the programming language and cannot be used for variable and function names. For instance, in Java program, words used for controlling the flow of programs such as *if* and *while* are reserved words.

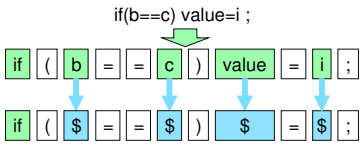

Fig. 2. Tokenization and Transformation

Figure 2 depicts an example of the tokenization of sour code. In this figure, green square represents alphabetic strings, and blue square represents converted variables. As you can see in this figure, a variable name is converted to the same token called $.

C. Step 3: Code Clone Detection

Generally, when a clone detection tool identifies code clones from large-scale software systems, it takes a huge amount of time or crashed due to lack of memory space [5]. To tackle this problem, CCFinderSW adopts n-gram for detecting clone detection with high speed. N-gram is a contiguous sequence of n words from a given sequence of string or words. It is frequently used to investigate the occurrence frequency of contiguous strings in the statistical field of natural linguistics or searching matching string.

Hereafter, the details of the process of code clone detection are explained. At first, CCFinderSW extracts n-gram from a sequence of transformed tokens, based on a threshold value set by the user. This threshold value represents that textsfC-CFinderSW only detects code clones that are more than n token length.

Suppose that user set up 4 as a threshold value when executing CCFinderSW and a sequence of transformed tokens is $sethesetheses$.

Based on this threshold value, CCFinderSW then creates a list of tuples (index, n-gram, hash value), where (1) **index** is an integer denoting the position of n-gram in the sequence (2) **n-gram** is a sequence contiguous of tokens with n token length based on a value defined by user and (3) **hash value** is a hash code for the n-gram.

The reason for using hash code is that comparison of numerical values is faster than comparison of strings. To calculate hash code, a hashcode() method in a class String was used.

Table III illustrates a created list from $sethesetheses$ with the 4 threshold value [1].

Next, it selects a unique hash value and its corresponding index from the list and then creates a unique list, which

[1]For the simplicity, this paper explains with character-based n-grams. Note that token-based n-grams are used in CCFinderSW.

TABLE II
AN EXAMPLE OF 4-GRAMS

order	4-gram
0	if($=
1	($==
2	$==$
3	==$)
4	=$)$
5	$)$+
6	)$=$
7	$=$;

contains unique hash value and its index. Table IV illustrates a unique list created from the Table III. As can be seen in the table, the hash value in the unique list contains more than two index values, because a hash value might appear more than twice.

Finally, code clones are identified based on the unique list. For that end, firstly, code clones whose token length is equal to the n threshold value are detected by using hash value and index value in the unique list. That is, the hash value with multiple index values implies that the same code appears in the multiple places, which indicates the existences of code clones. For example, in the first row of the list in the table IV, the index values of a (1234) hash value are $\{0, 5\}$. This means that the same hash value appearing in both 0th and 5th positions. This means that 4-grams appearing from 0th and 5th positions are equivalent. Therefore, 4 characters appearing from 0th to 3rd and 4 characters from the 5th to the 8th in the original string are detected as code clones. Furthermore, (2342) and (3421) hash value has $1, 6$ and $2, 7$ index value, respectively. This indicates that the code clones exist at 1th and 6th position and 2nd and 7th position.

Secondly, code clones with a maximum number of tokens are detected by checking the hash value of the next index for each element that is already detected clone clones. For example, code clones, whose hash value is (1234), appearing at the 0th and 5th position, as can be seen in the first row of the list in the table IV. To identify code clones with maximum number of tokens, CCFinderSW checks whether code clones also exist in the next position (1th and 6th) of detected codes. From table IV, you can see that the same hash value namely (2342) exist. ,

Since 4-gram starting at 0th and 5th positions are code clones and 4-gram starting from 1st and 6th positions are code clones, strings starting from 0th and 5th positions are detected as code clones with 5 tokens.

In this way, a new clone clones are identifying by determining hash values of the next index. This process is terminated by checking all hash values in the unique list.

As a result, Table II is a list of 4-grams.

For the simplicity, character-based N-grams are extracted, but token-based N-grams are used in the actual algorithm.

We define the target source code as $sethesetheses$. The threshold value of the number of tokens of the code clone is set to 4, and the detection target is a coincidence portion of 4 or more characters.

Since the threshold is 4, 4-gram is extracted from sethesetheses. Next, 4-grams are listed. The elements of 4-gram list include appearance, string, and hash value. The index of appearance is set such that the first 4-gram is 0, the next 4-gram is 1, and so on. The hash value is a hash of the string. This hashing is aimed at speeding up based on the fact that comparison of numerical values is faster than comparison of strings. In this way, a list of elements with a appearance, a string and a hash value is called list in order of appearance. Table III gives an example of a list in order of appearance.

However, the hash value shown in the figure is different from that implemented in CCFinderSW. Hashing in CCFinderSW is implemented using the hashcode method provided in class String. Also, in this explanation, it is assumed that collision of hash values does not occur. In other words, when the hash values are equal, the original character string is assumed to be equivalent.

From the set of hash value of this list in order of appearance, all unique hash values are selected. And CCFinderSW selects all appearance corresponding to each of the selected hash values. The correspondence of the appearance from the hash value is not only one-to-one, but it may be one-to-many. A list of elements with a set of hash values and a appearance is called a unique list. Table IV gives an example of a unique list.

Next, we searches for code clones. $\{0, 5\}$ of the first line of the created unique list is a set of appearances corresponding to the hash value (1234). When referring to the list in order of appearance for each element of this set, the same hash value appeared in those occurrence spots. In other words, values hashed from 4-grams appearing at appearance 0 and 5 are equivalent, so 4-grams are considered to be equivalent. Therefore, it is considered that the 4 characters from 0 to 3 and 4 characters from the 5th to the 8th in the original string are equivalent, and are code clone. Also, the set $\{1, 6\}$ of appearance corresponding to the hash value (2342) and the set $\{2, 7\}$ of appearance corresponding to the hash value (3421) are also the same as in the previous example, so sethesetheses has a set of clone sets. Hence, a code clone whose number of tokens is equal to the threshold value is detected.

Next, an algorithm for searching a code clone whose number of tokens is larger than the threshold value will be described. CCFinderSW checks the hash value of the next appearance for each element of the already detected clone set. For example, the tool adds 1 to each of the elements of clone set $\{0, 5\}$ to be $\{1, 6\}$, and look at the hash values corresponding to 1 and 6 in the list in order of appearance, so both match (2342). Since 0th 4-gram and 5th 4-gram match, and 1st 4-gram and 6th 4-gram match, strings starting at 0 and strings starting at 5 are code clones of 5-character. Thus the 4-character clone set $\{0, 5\}$ is rewritten to a 5-character clone set $\{0, 5\}$. In this way, a new clone set is searched by looking at the hash values of the next clone set in order of appearance. If this algorithm is performed on all the hash values of the unique list, the code

TABLE III
AN EXAMPLE OF A LIST IN ORDER OF APPEARANCE

{Appearance}	[String]	(Hash value)
{0}	[seth]	(1234)
{1}	[ethe]	(2342)
{2}	[thes]	(3421)
{3}	[hese]	(4212)
{4}	[eset]	(2123)
{5}	[seth]	(1234)
{6}	[ethe]	(2342)
{7}	[thes]	(3421)
{8}	[hese]	(4212)
{9}	[eses]	(2121)

TABLE IV
AN EXAMPLE OF A UNIQUE LIST

(Hash value)	{Appearance}
(1234)	{0,5}
(2342)	{1,6}
(3421)	{2,7}
(4212)	{3,8}
(2123)	{4}
(2121)	{9}

clone detection is over.

D. Step 4: Formatting

In this process, the output file is gereated by converting each location of detected code clones into line numbers on the original source files. For output file of CCFinderSW, two kinds of formats are available; One is the same output format as the CCFinder and others are the same output form as the CCFinderX.

IV. EVALUATION

We investigated the accuracy of CCFinderSW in terms of comment elimination.. We use Rosetta Code, a webpage that provides source code implemented in different programming languages for solving the same task[2]. Due to its abundant resources, Rosetta Code has been effectively used to compare languagesf concise, performance, and failure-proneness. In the evaluation, we extracted source code snapshot of November 18, 2015 from the *RosettaCode* Git repository[3].

We select a task named *Comments*[2], which contains programs with comments implemented in multiple program languages, in Rosetta Code. We then apply CCFinderSW to programs contained at *Comments* task. The details of the experiment are as follows.

We manually check programs in the *Comments* task and then choose 78 tasks as our studied subjects, because they are all implemented in the object-oriented manner.

1) We list the comment rules of each language that can be deleted by the proposed method used in Comments. Among them, we regarded comments that were possible by lexical analysis but can not be removed by the proposed method as impossible to remove.
2) We create 26 kinds of options. This option makes it possible to cover more language comment rules.
   In order to make it easier for users to set comment rules, we created a mechanism that allows comment rules to be set simply by giving an alphabet string. By limiting 26 types, we hope that it is easier to write comment rules as arguments when running the tool. For example, when executing the tool by giving argument adf on the command line, comment elimination is performed using the comment rule set to a, d, and f.
   Each option is shown in the Table V, a list of options for code elimination. The option of z concerns whether nesting of multi-line comments is allowed or not, so it is not included in the table. Numbers in the column of the category indicate the classification of comments, 1 is a line comment, 2 is a multi-line comment, 3 is a full line comment, 4 is a full multi-line comment, and 5 is a literal rule. The details of the results of applying the option to each language are described in the literature[3].

Using this option, we performed comment elimination to the 175 languages that implement Comments. For languages in which we could not be performed comment elimination, it was described as impossible in the option column. From this experiment, 166 languages out of 175 languages are possible by the proposed method, and 151 languages showed that it is possible to remove comments with 26 options. The reason why the proposed method can not cope with a language is that comments of a language can not be removed unless parsing and that a language had a difficult grammar. Then, the reason that 26 kinds of options can not support is that a comment rule is unique and there are too many comment rules.

Table VI is the number of languages used for each option in 166 languages.

V. RELATED WORK

So far, various tools have been developed for the detection of code clones in source code [6], [7], [2]. Most of the tools (e.g., CCFinderX [2]) support only a limited number of programming languages. Since NiCaD [8] detects code clones based on a TXL [9] grammar for each languages, it is able to support new programing language by writing a TXL grammar for the language. However, according to our experience in industry/university collaboration, it is difficult for most practitoners to write a programming language grammar and the parser for it correctly. Therefore, we developed CCFinderSW that is able to support a new programming language by specifying only a few options.

VI. CONCLUSION AND FUTURE WORK

In this paper, we introduced CCFinderSW, a tool that detects token-based codes clones for multiple program languages. Regarding token splitting, we did not set options with-

[2]http://rosettacode.org/wiki/Rosetta_Code
[3]https://github.com/acmeism/RosettaCode. Data
[2]http://rosettacode.org/wiki/Comments

[3]https://sites.google.com/site/yoshidaatnu/tableforAPSEC.pdf

TABLE V
26 OPTIONS (EXCLUDE Z)

Opt.	Cat.	Start	End
a	5	’	”
b	5	”	”
c	1	//	none
	2	/*	*/
	2	/+	+/
	2	<!–	– >
d	1	;	none
e	1	#	none
f	1	–	none
g	1	%	none
h	1	!	none
	1	# !	none
i	1	’	none
	1	‘	none
j	1	NB.	none
k	2	{	}
l	2	[	]
m	2	(	)
n	2	”	”

Opt.	Cat.	Start	End
o	2	(*	*)
	2	(:	:)
	2	(#	#)
p	2	#|	|#
	2	#=	=#
	2	#{	}#
	2	###	###
	2	#cs	#ce
	2	# ~	~#
q	2	{|	|}
	2	{-	-}
r	2	%{	%}
s	1	->	none
	1	–##	none
	2	–[[	]]
t	1	COMMENT	none
	1	IGNORELINE	none
	2	’COMMENT’	;

Opt.	Cat.	Start	End
u	1	©	none
	1	*>	none
	1	*|	none
	3	NOTE	none
	3	*	none
v	3	*	none
	3	C	none
	3	c	none
	3	:	none
w	2	%REM	%END REM
	1	REM	none
	1	rem	none
	1	Rem	none
x	4	==comment	=cut
	4	=begin	=end
	4	=pod	=cut
y	3	#;	none
	3	NIL	none
	3	###	none
	3	end.	none

TABLE VI
THE NUMBERS OF LANGUAGES FOR EACH OPTIONS

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
1	3	55	25	40	14	10	7	11	1	3	2	1	2	14	8	1	1	2	2	2	4	9	2	1	17

out literals. As a future work, we plan to prepare additional option for token splitting.

ACKNOWLEDGMENT

This work was supported by JSPS KAKENHI Grant Numbers JP25220003, JP15H06344 and JP16K16034.

REFERENCES

[1] S. Nanz and C. A. Furia, “A comparative study of programming languages in rosetta code,” in *Proceedings of the 37th International Conference on Software Engineering*, 2015, pp. 778–788.

[2] 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*, vol. 28, no. 7, pp. 654–670, 2002.

[3] A. Tosun, E. Shihab, and Y. Kamei, “On code reuse from stackoverflow: An exploratory study on android apps,” *Information and Software Technology*, vol. 88, pp. 148 – 158, 2017.

[4] T. Kamiya, “the archive of CCFinder Official Site,” 2005. [Online]. Available: http://www.ccfinder.net/

[5] S. Livieri, D. M. German, and K. Inoue, “A needle in the stack: efficient clone detection for huge collections of source code,” Osaka University, Tech. Rep., 2010.

[6] I. D. Baxter, A. Yahin, L. Moura, M. S. Anna, and L. Bier, “Clone detection using abstract syntax trees,” in *Proc. of ICSM 1998*, 1998, pp. 368–377.

[7] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, “Deckard: Scalable and accurate tree-based detection of code clones,” in *Proc. of ICSE 2007*, 2007, pp. 96–105.

[8] C. K. Roy and J. R. Cordy, “Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization,” in *Proc. of ICPC 2008*, 2008, pp. 172–181.

[9] J. R. Cordy, “The TXL source transformation language,” *Science of Computer Programming*, vol. 61, no. 3, pp. 190 – 210, 2006.