IJCSE International Journal of Computer Sciences and Engineering Open Access
Research Paper Vol-6, Special Issue-5, Jun 2018 E-ISSN: 2347-2693

Pragmatic Aspects of Token-based Technique in Detecting Source Code Duplicates

S. Bharti[1*], H. Singh[2]

[1, 2, *]Department of Computer Science, Guru Nanak Dev University, Amritsar, India

[*]*Corresponding Author: sarveshwar.dcsrsh@gndu.ac.in, Tel.: +91-9906129214*

Available online at: www.ijcseonline.org

***Abstract*—** Clone research community has described several techniques to detect code duplicates present in the code base, mainly categorized into four classes viz. textual or text-based techniques, lexical or token-based techniques, syntactic techniques (including tree-based and metrics-based approaches) and semantic techniques. Literature lists various clone detector tools based on each category capable of detecting clones in batch mode as well as in real-time development environment. But, most of the tools use tokens as their intermediate representation of the source code upon which clone detection algorithms are applied. Thus, this paper will focus on this token-based intermediate representation and its pragmatic aspects towards code duplication detection. By discussing the practical process of converting source code into tokens as an intermediate code representation and how code duplicates are detected, authors will put light on the obscured pros and cons of this token-based approach that will help researchers to select as well as implement, or reject this approach as an intermediate representation for their duplication detection algorithms.

***Keywords*—** Code Clone Detection, Clone Detection Techniques, Token-based Clone Detection Technique

I. Introduction

John Burdon Sanderson Haldane, a scientist with contribution to the fields of Biology and Biostatistics, was the first who invented the term clone, derived from the Ancient Greek word 'klon' (i.e. "twig"-meaning a small thin branch of a tree) and the process of creation of new plants from a twig is referred as a klon. Over the decades of research on software clones, several prominent authors presented their understanding of the term clone, but the widely utilized definition was given by Baxter *et al.* [1] stating "*a clone is a code fragment that [is] identical to another fragment*". The source code fragments can be identical based on syntactic or semantic similarity. Semantically similar code fragments are referred to as type 4 clones whereas syntactic clones are further classified into three types viz. type 1, type 2 and type 3 [2]. Clones are not inducted into the software system by its own, it is the software developer who is responsible for it. There might be various intentional or unintentional reasons behind the induction of clones [3]. Copy and paste activity [4] committed by the developer so as not to 're-invent the wheel' is the most eminent reason for the presence of clones in the software system. Thus, clones are sneaked in the software as a typical reuse approach. As per the empirical evaluation [5], it has been established that a particular software system may contain 9% to 17% of the cloned code. While offering few benefits in some cases, code clones are mostly detrimental. Thus, these duplicated code fragments should be removed from the software system and if possible reverted to be inducted into the software system.

Code clones can be detected by directly comparing source code as done in textual approaches, or may require source code to be converted into an intermediate representation to ameliorate the efficiency of the detection algorithms. Accordingly, researchers of the clone research community have devised various techniques [6] to identify the duplication in the source code viz. tracking clipboard operations, textual comparison, metrics comparison, token-based comparison, syntax comparison, PDG-based comparison, hash-based comparison etc. The token-based technique, a mostly used technique is the main concern of this paper.

The main motivation of this paper is that most of the real-world and efficient tools for detection of code clones are based on token-based approach, and, as the research papers mostly focus on their proposed approach and little information is discussed about the practical implementation of this token-based approach, thus this paper will present the detailed pragmatic aspect of the token-based approach and how this technique is used for detecting code duplicates in source code.

Section II will discuss few tools that are based on token-based clone detection and section III will discuss this token based clone detection approach in detail. Section IV will discuss how tokens are specified during lexical analysis and section V presents the procedure for recognition of token patterns. Then section VI will discuss how indexing data structures are used over token representation to identify clones. After it, attributes (section VII), efficiency (section VIII), key advantages (section IX) and limitations (section X) are discussed. Finally, this paper is concluded along with acknowledgments and references in support of this paper.

II. Related Literature

Over the decades of research on code clones, a number of different tools and techniques were proposed. Table 1 lists few tools from literature as an example of token-based techniques for clone detection.

Table 1. Token-based Clone Detection Tools

Author	Tool	Comparison Technique	Reference
Kamiya *et al.*	CCFinder	Suffix tree based token matching	[7]
B. Baker	Dup	Suffix tree based token matching	[8]
Li *et al.*	CP-Miner	Frequent subsequence mining technique	[9]

Kamiya *et al.* [7] came up with a tool CCFinder based on token-based technique. The lexer is used to divide each line of the source file under consideration into tokens, and then the concatenation of the tokens from all the source files is done to form a single sequence of tokens. Transformation is then done over token sequence and identifiers are replaced by special tokens. Finally, a sub-string matching algorithm based on suffix tree is applied to identify similar sub-sequences from the transformed sequence of tokens and then mapping back to the original source code is done.

Dup [8] is another token-based tool, utilizing the functionality of the lexer for tokenizing the source code and then comparing the suffix-tree created for each line without applying transformation as of CCFinder.

CP-Miner [9] is another clone detection tool based on token-based approach. It used frequent subsequence mining technique to identify tokenized statement sequence.

III. Token-Based Clone Detection Techniques

Area of research on software clones emerged out during early 1990's and over the decades of research with significant contributions from the researchers, this field of study on software clones has become a substantive contributor towards software quality improvement and reduced maintenance effort. Clone research community has invented number of different techniques and implemented them as tools to detect and manage clones. These clone detection techniques or more specifically these clone detection algorithms basically rely on intermediate code representations, which give rise to different types of clone detection techniques viz. tree-based, token-based, text-based, graph-based, metrics comparison or even hybrid techniques. Out of these techniques, the token-based technique is most widely used and is discussed in detail in this section.

The token-based approach was introduced in clone detection to improve the efficiency of the clone detection algorithms. This technique uses an intermediate code representation which is in the form of a stream of tokens. Tokens are basically an undividable sequence of characters of the programming language. Token-based clone detection tools identify clones by comparing these tokens rather than comparing text or other intermediate representations. Token-based comparison comprises of transforming the source code into the stream of tokens through lexical analysis and then scanning this stream of tokens for any similar subsequence of tokens. The similar token subsequences are mapped to the corresponding source code and reported as clones. A stream of tokens may be used in two ways to estimate duplication, first is to model this stream as a 'bag of words' for source code and second involves structure-aware clone detection. The process of conversion of source code (i.e. a sequence of characters) into the stream (or sequence) of tokens is referred to as tokenization, lexing, lexical analysis, or simply scanning, which is performed by a program called as scanner, lexer, or tokenizer.

Lexing basically involves two stages [10]:

- Scanning
- Evaluating

Scanning: It is the process of segmenting the input source code (i.e. input string) into the syntactical units termed as lexemes, and then categorizing them into token classes.

Evaluating: It refers to the conversion of lexemes into the processed values.

Literature also mentioned that lexical analyzers may also be specified as containing two processes [11]:

- Scanning
- Lexical analysis

Scanning: It performs densification of successive whitespaces into one and removes comments, so it does not require tokenization.

Lexical Analysis: It is a complex process of producing tokens from the output of the scanner.

Aho *et al.* [11] defined lexeme as "*a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token*". They defined token as "*a pair consisting of a token name and an optional attribute value*" and a pattern as "*a description of the form that the lexemes of the token may take*".

Table 2 lists informal description and some sample lexemes for some typical tokens

Table 2. Example of Some Typical Tokens. [11]

Token	Informal Description	Sample Lexeme
if	Characters I, f	if
else	Characters e, l, s, e	else
comparison	< or > or <= or >= or == or !=	<=, !=
id	Letter followed by letters and digits	Pi, score, D2
number	Any numeric constant	3.14159, 0, 6002e23
literal	Anything but ", surrounded by "'s	"core dumped"

For a better understanding of token being pair of token name and token value, table 3 presents some common token names and example token values.

Table 3. Some Common Tokens and Example Token Value. [10]

Token name	Example of token values
identifier	x, color, UP
keyword	if, while, return
separator	},), ;
operator	+, <, =
literal	true, 6.02e23, "music"
comment	//must be negative, /*Retrieves user data*/

Out of the total token names listed in table 3, we can say that tokens are mainly of three types, viz. identifiers i.e. variables, operators, and literals i.e. constants.

Consider the following c expression:

z = x + y * 2

Yielding the lexemes: **{ z, =, x, +, y, *, 2}**

And, corresponding tokens are:

{<id, 0>,<=>, <id, 1>, <+>, <id, 2>, <*>, <id, 3>}

Let us take another statement (Fortran statement [11]) as example to describe the concept of token names and associated values as:

E = M * C ** 2

The sequence of pairs of the above statement can be written as [11]:

<id, pointer to symbol table entry for E>
<assign_op>
<id, pointer to symbol table entry for M>
<mult_op>
<id, pointer to symbol table entry for C>
<exp_op>
<number, integer value 2>

In other terms, let us take an example program fragment [12]:

x = a
break
x = x
y = a

The token table for this code fragment would be (as shown in table 4):

Table 4. Token Table. [12]

Token	id	=	id	break	id	=	id	id	=	id	_
Index	1	2	3	4	5	6	7	8	9	10	11

IV. Specification of tokens

Lexeme patterns are specified by an important notation referred to as regular expressions. To specify patterns that we actually need for tokens, regular expressions are very effective despite not expressing all the possible patterns. Context-free grammars are more powerful than regular expressions but context-free grammars cannot express every construct expressed by regular expressions and vice-versa. In the specification of tokens, the frequently used terms are defined as:

Symbol: It is the basic building block and can be a letter, digit etc.

Alphabet: Any finite set of symbols is referred to as alphabet e.g. letters, digits and punctuation.

String: Aho *et al.* [11] defines string over an alphabet as "*a finite sequence of symbols drawn from that alphabet*"

Language: A language is "*any countable set of strings over some fixed alphabet*" [11].

Operation on Languages: Concatenation, union, and closure are the most significant operations on languages during lexical analysis.

Regular expressions are usually used to describe all the languages that can be built up from operators employed to the symbols of some alphabet. The language is called regular set if it can be defined by a regular expression. For example, to specify (i.e. to describe) the set of valid identifiers (i.e. a string of digits, letters, and underscore) for C language via representing letter_ for letter or underscore and digit for digit, the language would be [11]:

letter_ (letter_ | digit)*

And, the regular definition would be:

letter_ → A | B | … | Z | a | b | … | z | _
digit → 0 | 1 | … | 9
id → letter_ (letter_ | digit)*

V. Recognition of tokens

The previous section described how to represent patterns by using regular expressions. This section will discuss how input string is examined to find the matching patterns. For describing lexical analyzers or any pattern processing software, the notation used is a regular expression, but to implement it requires simulation of deterministic finite automata (DFA), sometimes non-deterministic finite automata (NFA). To construct lexical analyzers, an intermediate step is a conversion of patterns (i.e. regular expression patterns) to transition diagrams. Finite automata are the formalism for transition and similar to transition diagrams that act as recognizers for the input string. Nondeterministic finite automata and deterministic finite automata are two flavors for finite automata. The literature mentions algorithms for simulating DFA and NFA for recognizing strings as presented below:

Algorithm 1: Simulating a DFA [11]

Input: An input string x terminated by an end-of–character eof, DFA D with start state s_0, accepting states F, and transaction function move

Output: "yes" if D accepts x, otherwise "no"

1. **s = s_0;**
2. **c = nextChar();**
3. **while (c != eof) {**
4. **s = move(s, c);**
5. **c = nextChar();**
6. **}**
7. **if (s is in F) return "yes";**
8. **else return "no";**

Algorithm 2: Simulating a NFA [11]

Input: An input string x terminated by an end-of –character eof, NFA N with start state s_0, accepting states F, and transaction function move

Output: "yes" if N accepts x, otherwise "no"

1. **S = ϵ-closure(s_0);**
2. **c = nextChar();** // returns next character of the input string
3. **while (c != eof) {**
4. **S = ϵ-closure(move(S, c));**
5. **c = nextChar();**
6. **}**
7. **If (S ∩ F != Ø) return "yes";**
8. **else return "no";**

To recognize the collection of keywords or a single keyword, there are Aho-Corasick algorithm and KMP algorithm respectively. The KMP algorithm to test whether a single keyword b_1 b_2 ……b_n is contained in a string a_1 a_2 …..a_m as a substring is presented below:

Algorithm 3: KMP algorithm for recognizing single keyword in a string [11]

1. **s = 0;**
2. **for (i = 1; i <= m; i++){**
3. **while (s > 0 && a_i != b_s + 1) s = f(s);** // f(s) is a failure function
4. **if (a_i == b_s + 1) s = s + 1;**
5. **if (s == n) returns "yes";**
6. **}return "no"**

VI. Implementation of token-based clone Detection Technique

Majority of the effective token-based code clone detection techniques are fundamentally established on suffix trees that were originally used for efficient string search. There are other alternatives for suffix trees like suffix array, compressed suffix trees etc. Brenda Baker extended this algorithm to parameterized string for code clone detection. This approach has an advantage over the original string search approach of finding cloned token sequences containing renaming of parameters.

Implementation of token-based clone detection involves employment of suffix tree (or any other indexing technique like compressed suffix trees, suffix array etc.) construction to the tokens of the source code under consideration. Thus, this technique involves the conversion of source code into the intermediate representation of tokens, then on the tokens derived from the program, the index construction algorithm is applied and another representation is extracted. After the construction of suffix trees or any other indexing data structure, clones i.e. similar token sequences are identified.

For example, the suffix tree for the tokens of table 4 is represented in figure 1 as:

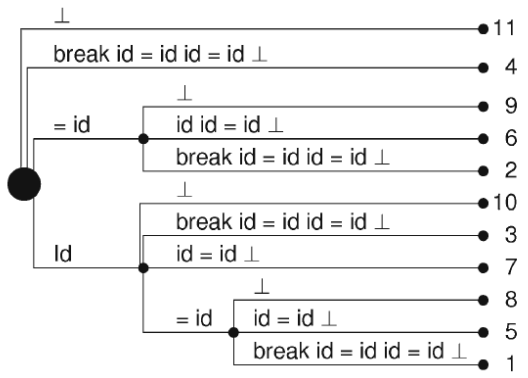

Figure 1. Suffix Tree. [12]

To detect clones, another representation of suffix tree for parameterized strings is as shown below in figure 2.

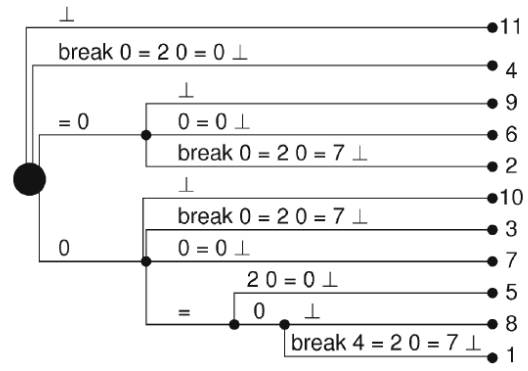

Figure 2. Suffix Tree for p-string. [12]

All the suffixes of a string are represented using suffix tree as a trie. The suffix is represented in suffix tree through a root to leaf path, where edges represent non-empty sub-strings and paths with common prefixes share the tree edges. In suffix trees, the clones are identified as an inner tree node of the trie representation.

VII. Attributes of The Token-Based Clone Detection Techniques

Roy and Cordy [3] presented 11 properties based upon which various clone detection techniques can be efficiently analyzed, viz. Source Transformation/Normalization, Source Representation, Comparison, Granularity, Comparison Algorithm, Computational Complexity, Clone Similarity, Granularity, Language Independency, Output/Groups of Clones, Clone Refactoring and Language Paradigm. Table 5 presents the description with reference to these dimensions for the token-based approach.

Table 5. Properties of Token-based Clone Detection Techniques

Dimension	Token-Based Clone Detection Technique
Transformation /Normalization	Source code is transformed into tokens through lexical analysis omitting whitespace & comments
Source Representation	Intermediate representation of source code is a sequence of tokens
Comparison Granularity	Tokens are compared to identify the similarity between code fragments
Computational Complexity	Overall complexity of algorithm based on tokens as intermediate representation is mainly Linear
Comparison Algorithm	Suffix Tree, Array, data mining, IR, Sequence matching, etc.
Clone Similarity	Exact Match, Renamed match, Near miss
Clone Granularity	Granularity can be Free or Fixed
Language Independency	Lexer for the language under consideration is needed to convert the source code written in any language to the stream of tokens
Output/Groups of Clones	Output of the token-based clone detection approach is Clone pair or Clone class
Clone Refactoring	Post processing is needed involving mapping of tokens back to source code to refactor the code
Language Paradigm	This approach can be applied to language paradigms like Procedural, OOP

VIII. Efficiency of token-based techniques

To emulate the efficiency of token-based techniques, several frequently used evaluation metrics are Precision, Recall, and F-measure. Precision tells us about the relevant instances out of all the retrieved clones whereas recall points to the relevant instances detected by the algorithm out of all the clones present in the code base. F-measure depicts the harmonic mean of precision and recall. Table 6 presents precision and recall calculated by researchers (few studies from literature) for token-based approaches.

Table 6. Efficiency of Token-based Techniques

Author	Recall	Precision	Reference
Bellon *et al.*	High	Low	[2]
Bailey and Burd	High	Low	[13]
Baxter *et al.*	Low	High	[1]

IX. Key Advantages of Token-based Approach over other Approaches

Token-based approach for detecting code clones has various advantages over other approaches, but few key advantages are as:

- Token-based technique has linear complexity in both time and space, thus can scale for large software systems [12]
- Even syntactically incorrect and incomplete code can be converted into stream of tokens as parsing is not necessary [12]
- Compared to other approaches, this approach can be easily adjustable to any new language [14]
- This approach is independent of the layout [12]
- Token-based (lexical) approaches are more robust than text-based approaches over minor changes in code such as renaming, spacing, and formatting [15]
- Writing lexical analyzer require less effort as compared to the developing syntactic analyzer [12]

X. Limitations of Token-based Approach

In spite of having dominant advantages over other techniques, it also has some limitations as listed below:

- Token-based code clone detection approach may detect clones as per lexical view but may not be clones from developer point of view
- To detect syntactic clones, token-based approach requires post-processing of sequence identified
- Separation of non-parameter and parameter tokens is another limitation of token-based techniques that cause false negatives to be identified during code clone detection

The limitations of token-based approaches are not permanent, but require extra effort to resolve, thus can still act as a better technique for detection clones with better precision and recall than other approaches.

XI. Conclusion

This paper discussed token-based intermediate representation as code representation and its pragmatic aspects towards code clone detection. The detailed discussion is presented on how tokens are produced, how tokens are specified, how token recognition is done and how clone detection is performed using token-based techniques. Then attributes, efficiency, key advantages, and limitations are discussed. This paper specified how indexing data structures are used in clone matching based on token-based code representation. Talking about the process of tokenization i.e. practical procedure of converting source code into tokens as an intermediate code representation and how code duplicates are discovered by utilizing various indexing data structures applied over token representation will help researchers in selecting, implementing or rejecting this technique as an intermediate representation for their duplication detection algorithms.

XII. Acknowledgment

Authors would like to acknowledge the financial, scholastic and infrastructural support from UGC and Department of Computer Science, Guru Nanak Dev University, Amritsar.

References

[1] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant' Anna, and Lorraine Bier, "Clone Detection Using Abstract Syntax Tree," in Proceedings of 14th International Conference on Software Maintenance(ICSM'98), Bethesda, Mayland, 1998, pp. 368 - 377.

[2] Stefan Bellon, Rainer Koschke, Giuliano Antoniol, Jens Krinke, and Ettore Merlo, "Comparision and Evaluation of Clone Detection Tools," IEEE Transaction on Software Engineering, vol. 33, no. 9, pp. 577 - 591, 2007.

[3] Chanchal K. Roy and James R. Cordy, "A Survey on Software Clone Detection Research," Queen's University, Kingston, Technical Report 2007-541, 2007.

[4] Miryung Kim, Lawrence Bergman, Tessa Lau, and David Notkin, "An Ethnographic Study of Copy and Paste Programming Practices in OOPL," in Proceedings of the 2004 International Symposium on Empirical Software Engineering (ISESE'04), Redondo Beach, CA, USA, USA, 2004.

[5] Minhaz F. Zibran, Ripon K. Saha, Muhammad Asaduzzaman, and Chanchal K. Roy, "Analysing and Forecasting Near-miss Clones in Evolving Software: An Empirical Study," in Proceedings of the 16th IEEE International Conference on Engineering of Complex Computer Systems, Las Vegas, USA, 2011, pp. 295-304.

[6] M. F. Zibran and Chanchal Kumar Roy, "The Road to Software Clone Management: A Survey," Department of Computer Science, University of Saskatchewan, Canada, Technical Report 2012.

[7] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro 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, July 2002.

[8] Brenda Baker, "On Finding Duplication and Near Duplication in Large Software Systems," in Proceedings of the 2nd Working Conference on Reverse Engineering (WCRE'95), 1995, pp. 86 - 95.

[9] Zhenmin Li, Shan mar, Yuanyuan ZohuLu, and Suvda Myag, "CP-Miner: Finding Copy Paste and Related Bugs in Large Scale Software Code," IEEE Transaction on Software Engineering, vol. 32, no. 3, pp. 176 - 192, March 2006.

[10] Wikipedia.[Online]. https://en.wikipedia.org/wiki/Lexical_analysis

[11] Alfred V. Aho, Monica S. Lam, and Jeffrey D. Ullman Ravi Sethi, Compilers: Principles, Techniques, and Tools, 2nd ed.: Pearson.

[12] Raimer Falke, Pierre Frenzel, and Rainer Koschke, "Empirical Evaluation of Clone Detection using Syntax Suffix Trees," Empirical Software Engineering, vol. 13, no. 6, pp. 601 - 643, July 2008.

[13] Elizabeth Burd and John Bailey, "Evaluating Clone Detection Tools for Use during Preventative Maintenance," in Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation (SCAM '02), Montreal, Canada, 2002, pp. 36-43.

[14] M. Rieger, "Effective Clone Detection without Language Barriers," University of Bern, Switzerland, Dissertation 2005.

[15] Chanchal Kumar Roy, James Cordy, and Rainer Koschke, "Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Quantitative Approach," Science of Computer Programming, vol. 74, no. 7, pp. 470 - 495, March 2009.

Authors Profile

Mr. Sarveshwar Bharti is presently working at the Department of Computer Science, Guru Nanak Dev University, Amritsar, India, as a Ph.D. Research Fellow (Senior Research Fellow). He has received his Master of Computer Applications (MCA) degree from University of Jammu, Jammu, India. He is a Software Engineering Researcher with research interests including Software Clones, Integrated Clone Management, and Clone Management Plug-in.

Dr. Hardeep Singh, Ph.D., is a Professor and Head at the Department of Computer Science, Guru Nanak Dev University, Amritsar, India. His research interests lie within Software Engineering and Information Systems. He has been awarded with various prestigious awards including Dewang Mehta Award for best Professor in Computer Engineering, ISTE Award for Best Teacher in Computer Science and Rotract International Award for best Teacher.

