Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 171 (2020) 2522–2531

Procedia Computer Science

www.elsevier.com/locate/procedia

Third International Conference on Computing and Network Communications (CoCoNet'19)

Automatic Source Code Documentation using Code Summarization Technique of NLP

Menaka Pushpa Arthur*

Associate Professor, Department of Information Technology, St.Joseph's Institute of Technology, Chennai 600119, India

Abstract

Source code documentation is an important process for software project maintenance and management. The documentation process always consumes a lot of time and effort from human experts. The software project document should be concise and clear without any ambiguity. Existing source code documentation tools like JavaDoc are very limited in the market. Also, tools can identify only the predefined methods of corresponding programming languages. In this paper, a novel system is proposed to automate the source code documentation process for C programming language using the source code summarization technique of NLP. The core component of this proposed system i.e., Software Word Usage Model (SWUM) build using Context-Free Grammars and NLP preprocessing techniques. This system can successfully generate the documentation for a C program along with predefined and user-defined methods using Natural Language Generation technique. This proposed system can document a program in two major formats; method-based, abstract level and statement-based, detailed level. The proposed system efficiency is evaluated by comparing system-generated source code documentation with an expert generated documentation. Results obtained from that comparison shows that the proposed system can give better performance for small and medium-size software projects.

Keywords: Source code documentation; CFG; NLP; code summarization; SWUM; etc

* Corresponding author. Tel.: +91-9994133025.
E-mail address: menaka_engg@yahoo.com, menaka.amp@gmail.com

1. Introduction

Documentation of the software project is mostly written by the human in software industries. This process is one of the important steps in the software life cycle model. Better documentation will reduce the cost of project maintenance and update processes. Software project Documentation maintains detail about a requirement, analysis, coding, testing and installation procedures of the software development cycle. This research work concentrates only on automatic documentation generation for coding. Coding documentation should be concise and clear without any ambiguity for developers of the maintenance process. Usually, documentation consumes a lot of effort and time from a human. The number of documentation pages per KiloCode is one among the metric which is used for estimating software project cost. Efficient documentation will be a very helpful part of the version update of the software project. The documentation development time and cost can be reduced by this proposed automatic source code documentation system using NLP techniques. This proposed system can generate documentation for C programs due to its simple syntax set among other object-oriented programming languages like Java. This research work develops a prototype model for an automatic source code documentation process by following the simple syntax-based, C-lite programming language compiler. Existing source code tools are few and have limited functionalities like JavaDoc, it generates documentation only for predefined methods of Java by extracting information from the Java programming language's official documentation. JavaDoc cannot automatically generate source code documentation for user-defined methods. Depends on user comments JavaDoc can generate documentation for user-defined methods.

In this paper, the proposed system can automatically generate the documentation for C program by Natural Learning Processing techniques. Two major documentation types are generated by this system; (i) In method-based documentation, methods like user-defined and pre-defined are automatically documented using extended method call graph technique. In this, user-defined methods are documented by the Natural Language Generation (NLG) grammars whereas pre-defined methods can be documented with the help of documentation of C programming language's compiler (ii) In the statement-based documentation, each program lines are documented along with methods if any exist in the given program. The proposed system uses C program as an input in order to generate documentation as an output. Program levels of complexity vary with the existence of user or pre-defined methods in this. Context-Free Grammars are formed for the syntax of C programming language. This set of rules can identify types of program statements such as expression, condition, declaration and function definitions, etc. Then, each statement type is converted into line documentation with necessary identifiers using NLG technique. Software Word Usage Model (SWUM) of this system is developed by a unique technique that acts as the core module for the software code documentation process.

The organization of this paper is as follows: The related work is discussed in Section II. In Section III, the proposed automatic source code documentation system is well described. In Section IV, the simulation results are discussed to demonstrate the proposed system's efficiency. Finally, conclusions and future enhancements of this work are discussed in Section V.

2. Related Works

The existing research works in software source code interpretation using NLP techniques are limited. G.Sridhara et al. [3, 4] and E. Hill et al. [7] have been initiated and contributed a lot of research works in this domain. P. W. McBurney et al. [1] proposed a system to generate source code summarization of context for Java methods. This system generates a summarization for method signatures. Method signatures are interpreted by the NLP techniques and summarization is generated by NLG technique. Hence, this system requests a strict way to define a method signature with an appropriate name for the method and its parameters and then the return value of this method. The method signature is parsed and tagged then identified the meaning of a method from its name by considering this as a verb in POS tagging process. The actual intention of this system is to replace the well-known JavaDoc for method documentation in Java. The limitation of this P. W. McBurney et al. [1] system is signature-based code summarization. The quality of the summarization depends only on the signature of the method. If the method signature is general or irrelevant to functions task then we cannot expect meaningful summarization from this system. JavaDoc converts user given comments and Java programming language documentation for the corresponding method into documentation for both user-defined and pre-defined Java methods. Whereas P. W. McBurney et al. [1] system converts programming language statements into natural language description for Java methods. This system cannot interpret the syntax of programming language other than the method signature.

S. Badihi et al. [2] proposed a system for automatic code summarization for Java programs using Eclipse-plug-in along with web-based crowd summaries. They have designed such a system using NLP and NLG techniques for interpreting programming sentences and code automatic summarization. Program summary from experts are collected using crowd sourcing. Crowd sourcing is an additional overhead process in this project also it reduces the quality of the expert's code summarization content. An important method is identified based on its frequency in the program. Method's linguistics such as action, arguments and themes are identified using NLP techniques that used to generate a summary for the concern Java method. This method expects the user to strictly follow naming conventions when they write the program. Irrelevant and meaningless naming for methods and their arguments in general identifier reduces the quality of this system. G.Sridhara et al. [3] generate code summary for Java method from its method call and signature using the combination of the techniques NLP and NLG. Lexicalization used to identify variables from the methods. This system identifies the S_Units from the method that contains starting, ending lines and important lines such as a loop from the method, in order to know the return type of the method. Software Word Usage Model has been build for software analysis using NLP by Pollock.L et al. [4]. Parsing the program sentence and analyzing method signature has been done via SWUM is explained well in this paper. Possible applications of software analysis using their system were also listed out in this work. They have mainly highlighted identifier extraction, POS tagging and abbreviation expansion in order to improve the accuracy of the system. This system generates a code summary for methods mainly from its signature.

J. Fowkes et al. [5] designed an unsupervised, extractive source code summarization system using an auto-folding method. This system creates Abstract Syntax Tree (AST) for source code written in Java. They have formulated the summarization problem mathematically and code blocks are identified to generate a program summary. Finally, the system generated summary is compared with the summary generated by Javadoc and then find out the best code summary from this analysis. They didn't use NLP technique and claimed that NLP is the extension of their research work. Few more systems have been designed with the help of Natural Language Processing to interpret the source code that is not for generating a code summary. Francu et al. [6] proposed a system to automatically generate executable Java source code from its system requirement using NLP. The system requirement is represented in the form of UML. E.Hill et al. [7] designed a system that finds an exact position in the large size software for maintenance and reuse within minimum search time. Identifier split and POS tagging are used to identify the important keywords in the software code. Improved NL query can be generated using the set of extracted important keywords. Lei et al. [8] introduced a reverse engineering process in software engineering. They have designed a system to automatically generate C++ parser from English language specification about the input files of the software program.

3. Automatic Source Code Documentation System

This section presents an architecture and working procedure of the proposed automatic source code documentation system using the code summarization technique of Natural Language Processing. The system accepts C program as an input with the option of the existence of pre-defined or/and user-defined functions. This proposed system can automatically generate concise source code documentation in a natural language without any human intervention. Two major types of source code summarization are generated by this system; (i) program or source code documentation and (ii) method documentation. The proposed system contains three major modules; (i) Preprocessing of C program (ii) Building a Software Word Usage Model (SWUM) (iii) Important methods extraction and (iv) Source code documentation generation using NLG for an entire C program or only for extracted important methods of a given program as per user choice. The detailed architecture of the proposed system is shown in Figure 1 with sub-components. The working procedure of this system is given in Figure 3 in the form of an algorithm. This system can generate documentation for each and every line of C program. Existing systems or tools can generate documentation only for the pre-defined or user-defined methods of the program and it cannot automatically generate documentation for the entire program code.

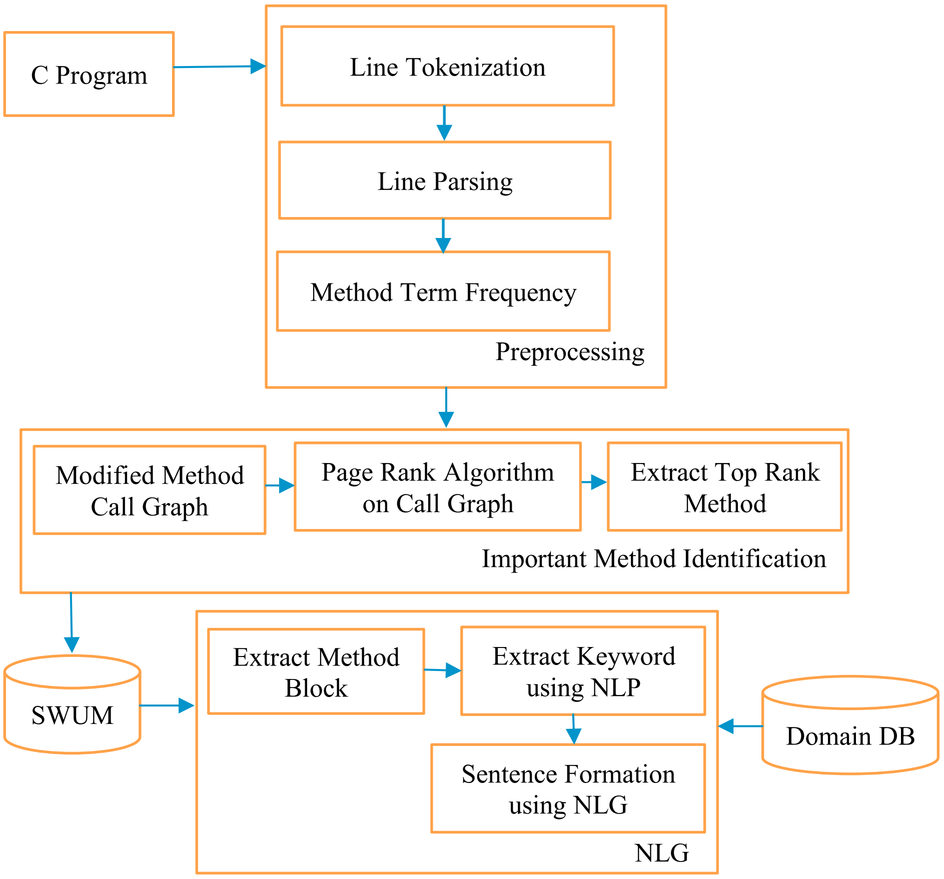
C Program
Line Tokenization
Line Parsing
Method Term Frequency
Preprocessing
Modified Method Call Graph
Page Rank Algorithm on Call Graph
Extract Top Rank Method
Important Method Identification
SWUM
Extract Method Block
Extract Keyword using NLP
Sentence Formation using NLG
NLG
Domain DB

Fig. 1. The Architecture of Automatic Source Code Documentation System.

3.1. Program Preprocessing

This module generates an output from the given C program and that output will be utilized to build a software word usage model; extract important methods from the program and generate the source code documentation for the program lines. Program statements or lines are tokenized using newline delimiter in a regular expression. The total number of program lines and then the set which contains the program line is obtained from the line-based program tokenization process. Then, POS tagging is applied to each retrieved program lines from the generated line set. From this, sentence components like a verb, noun, pronoun, etc. are identified for each line except the program lines that contain only the keywords of the programming language. Parsing of program lines using space delimiter in the regular expression can identify the type of program statements/lines such as the declaration, control, expression, assignment, etc. Program lines are parsed using regular expressions with POS tagging and then identified by CFG rules. The line parsing process is explained in the next section as it is with the software word usage model building phase. The frequency of method call is identified by the term-frequency technique.

3.2. Software Word Usage Model

SWUM is the core component of the code summarization technique in NLP. The way to build SWUM is varying depends on its purpose. This work creates SWUM for the source program by considering individual program lines with its type, parsing detail, type of the methods (pre-defined or user-defined) and method frequency, etc. Parsing of program lines is carried out based on the syntax of programming language that is in the form of Context-Free Grammars. The syntax of C programming language is defined in the form of CFG rules. The sample CFG rules of this proposed system are given in Figure 2. Regular expression technique maps the program lines with a set of CFG programming language syntax rules in order to identify the type of that program statement. In this work, syntax rules are formed based on the syntax of the C-Lite compiler. The syntax rules of the C-Lite compiler is very light and compact than the C compiler. Due to this reason, this work adopts the syntax rules of the C-Lite compiler for developing a prototype model for an automatic source code documentation system. In future work, this prototype system model will be enhanced and modified with respect to other high-end programming language compilers like object-oriented programming languages and task-specific programming languages, etc. The information about the program statement type gives the actual purpose of the corresponding statement in the logic of the problem-solving. In the next step, POS applied to each line and find the statement components like verb, noun, adverb, etc. Those components are given as one among the input for NLG process to generate the descriptive program line documentation. This line POS information acts as a basic building block for sentence formation in the documentation generation phase. Methods are identified separately along with its frequency and the relationship among methods like method interaction type, interaction frequency, the importance/rank of the method and the order of method call and so on. This information is the main input for the Modified Method Call Graph phase of the system.

```
Statement → Stat | Control | Library | Method | Key
Stat → Expr1 | Declaration1 | Key Integer Key | Key Identifier Key
Expr1 → Expr ';'
Declaration1 → Declaration ';'
Declaration → DataType Identifier | DataType Identifier Key Identifier | DataType Identifier '=' Integer | DataType
Identifier '=' Identifier | DataType Identifier '=' Expr | Identifier '=' Identifier | Identifier '=' Integer | Identifier '='
Expr | Identifier '=' MethodCall1 | DataType Identifier '=' MethodCall1
Expr → Identifier Op Identifier | Identifier Op Integer | Integer Op Identifier | Integer Op Integer
Library → Key ';'
Method → MethodDecl1 | MethodCall1 | MethodDefn
MethodDec1 → MethodDec ';'
MethodCall1 → MethodCall ';'
MethodDec → DataType Method
MethodCall → Method
MethodDef → DataType Method Key Statement Key
Method → Identifier Key ParamList Key
ParamList → DataType Identifier | DataType Identifier Key ParamList | Identifier | Identifier Key ParamList |
Integer | Integer Key ParamList
Control → Key Condition Key Statement Key | Key Condition Key Statement Key Statement | Key Condition Key
Statement Key Key Statement Key
DataTypes → 'int' | 'float' | 'char' | 'char[' Integer ']' | 'double'
Key → Keywords | Keywords Key
Keywords → 'printf' | 'scanf' | '#include<stdioh.h>' | '#include<math.h>' | '#include<string.h>' |'#include<conio.h>'
| 'return' | 'main' | 'int' | 'void' | 'String' | '{' | '}' | ';' | ',' | '>' | '(' | ')' | 'switch' | 'case' | 'break' | 'do' | 'if' | 'else' |
'while' | 'for'
Identifier → Alphabet | Identifier Integer | Alphabet Identifier | Alphabet Identifier Alphabet
Alphabet → 'a' | 'b' | 'c' | 'd' | 'e' | 'f'|....|'z'|'A'|'B'|.....|'Z'
Integer → Digit | Digit Integer
Digit → '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
Op → '+' | '-' | '*' | '/' | '**' | '%'
```

Fig 2. Context-Free Grammar for Programming Language Syntax

3.3. Modified Method Call Graph

Method call graph explains the way of calling a method from other methods. It gives the relationship between methods. The order of method calling given by the method call graph uses for proper documentation about function calling within the program or project. This project modified the method call graph by including the detail about the nature of methods such as pre-defined or used-defined or from other program files etc. An arc of this modified graph carries a weight that indicates the calling frequency of the corresponding method. The methods which all have a direct call from the main method are marked as important methods. And, based on the total number of lines of code of the method also used to identify the importance of that method with respect to a software project. The methods which have large LOC are considered as important methods. As per the input program is given in Figure 4(a), method main() contains 19 lines whereas other methods addition and subtraction has a single line. Form this, we cannot claim the method main as the important method of the given program based on the maximum LOC. In the modified method call graph always consider the main () as the highest priority important method of any program regardless of its LOC. The actual task of the method is also considered to identify the important methods in the program. This phase needs further research to calculate the exact weight of each method of the program. Important methods are only identified by the modified method call graph and extracted for the method documentation process.

3.4. Code Documentation using NLG

The source code documentation is carried out in two ways; method-description (short documentation) and line-wise description (detailed documentation). A set of natural language generation rules are formed to generate sentences either for the entire program or extracted important methods of the given program. The rules set is majorly classified as two, based on the input of this phase. The tag value and type of the program sentence play a major role to form its corresponding natural language sentence. The detailed documentation may help in the reverse engineering process of software engineering in the future.

```
Input    : C Program Code and CFG, NLG rules
Output   : Source Code Documentation
1.  procedure preprocessing (program p, CFG rules)
2.  split program lines using newline delimiter
3.  create an array of program lines 'l' by line tokenization
4.  for each line / element of 'l'
5.  find line type using CFG rules
6.  store line type in 'type'
7.  for each method type of statement
8.  find the frequency of method calling and store it in 'freq'
9.  create software word usage model 'sw'
10. return sw, l, type and freq

11. procedure modified_methodcall_graph (number_of_methods, method, freq)
12. create 'n' vertices that is equal to number_of_methods
13. assign weight 'w' for each m1 and m2 link; if method 'm1' calls a method 'm2'
14. 'w' is assigned as 'freq'
15. Other link weights 'w' set as '0'
16. find the page rank method to extract 'm' with high 'freq'
17. return top rank methods 'm'

18. procedure code_summary(program p, CFG rules, NLG rules1, swum sw)
19. call preprocessing (p, rules)
20. for each line 'l' in 'p'
21. call part_of_speech tagging procedure
22. for each 'l' and its 'tag'
23. find 'verb' and 'noun' and other components
24. create a sentence using 'components' and 'NLG' rules
25. return sentence / documentation of each 'l'
```

```
26. procedure method_summary(program p, CFG rules, NLG rules1, swum sw)
27. tm = call modified_methodcall_graph(p, rules)
28. extract top ranked 'tm' method from 'p'
29. call code_summary(tm, rules, rules1)
30. return important method documentation

31. procedure main (program p, CFG rules, NLG rules1)
32. sw = call preprocessing (p, rules)
33. call code_summary(p, rules, rules1,sw)
34. call method_summary(p, rules, rules1,sw)
35. create documentation and print
```

Fig 3. Algorithm of the Proposed System

4. Results and Discussion

```
#include <stdio.h>
int my_add(int x, int y)
{
   return (x + y);
}
float my_sub(float x, float y)
{
   return (x-y);
}
int main()
{
   int a, b, c;
   float a1, b1, c1;
   printf("Enter a : ");
   scanf("%d", &a);
   printf("Enter b : ");
   scanf("%d", &b);
   c = my_add(a,b);

   printf("Result of Addition : %d", c);

   printf("Enter a1 : ");
   scanf("%f", &a1);
   printf("Enter b1 : ");
   scanf("%f", &b1);
   c1 = my_sub(a1,b1);

   printf("Result of Addition : %f", c1);

   return 0;
}
```

Fig 4(a). Source Program

```
Library
Method → MethodDef
BlockStart
Expression → Addition → Return
BlockEnd
Method → MethodDef
BlockStart
Expression → Subtraction → Return
BlockEnd
Method → MethodDef → Predefined → Main
BlockStart
Identifier → a,b,c → Integer
Identifier → a1, b1, c1 → Float
Method → MethodCall → Predefined → Print
Method → MethodCall → Predefined → Read → a
Method → MethodCall → Predefined → Print
Method → MethodCall → Predefined → Read → b
Method → MethodCall → UserDef → my_add → a,b
            Return → c
Method → MethodCall → Predefined → Print → c

Method → MethodCall → Predefined → Print
Method → MethodCall → Predefined → Read → a1
Method → MethodCall → Predefined → Print
Method → MethodCall → Predefined → Read → b1
Method → MethodCall → UserDef → my_sub → a1,b1
            Return → c1
Method → MethodCall → Predefined → Print → c1

Keyword → Return → 0
BlockEnd
```

Fig. 4(b) Parser – Sentence Type Identification

The proposed system is implemented by Python with necessary Natural Language Processing and Natural Language Generation packages. The sample input C program is given in Figure 4(a) that will help to identify the output generated by the proposed system along with intermediate results such as output of the sentence parser and modified method call graph. This program does not have any developers' comments. Without the developer's comments, the proposed system can generate the documentation for any source program. In contrast, Javadoc requests code comments to generate the method summary. Figure 4(a) shows the C program that performs the addition of two integers and subtraction of two float numbers. User-defined methods of this program are *my_add()* and *my_sub()* with two arguments in the argument list along with the program's entry point. The program statement type is identified by the parser and SWUM modules of the system. The program statement type generated by the Parser and SWUM is given in Figure 4(b) for each and every line of the source code including header file-inclusion. From this parser, user-defined and pre-defined method sections are extracted and then analyzed to generate the method call graph of the given program. This proposed system has a modified method call graph component that lists the detail such as important methods in the program, LOC, interaction type, interaction direction, arguments of the interaction and purpose of the method. The output of the method call graph module for the given program is given in Figure 5.

```
Modified Method Call Graph Information:

int main () → my_add(a,b)
Count: 1
Parameter from int main () → my_add(a,b): 2 integer
Parameter from my_add(a,b) → int main (): 1 integer
Task: addition

int main () → my_sub(a1,b1)
Count: 1
Parameter from int main () → my_sub(a1,b1): 2 float
Parameter from my_sub(a1,b1) → int main (): 1 float
Task: subtraction

Important Methods with LOC:
int main () : 19
my_add(a,b) : 1
my_sub(a1,b1) : 1
```

Fig 5. The Output of Modified Method Call Graph for the Program given in Fig 4(a)

```
User Defined Method "my_add" for Addition of "a"
(integer), "b" (integer).
Return result (integer).
Store in "c" (integer).

User Defined Method "my_sub" for Subtraction of
"a1" (float) and "b1" (float).
Return result (float).
store in "c1" (float).
```

Fig 6(a) Source Code Documentation for Methods

```
Library "stdio" included
User Defined Method "my_add" for Addition of "a"
(integer), "b" (integer). Return result (integer).
User Defined Method "my_sub" for Subtraction of "a1"
(float) and "b1" (float). Return result (float).

Main Starts
Declaring Identifiers a(int), b(int), c(int),
a1(float),b1(float),c1(float)
Read a(int), b(int), a1(float), b1(float)
Call Method my_add with a(int), b(int). Save Result in c
(int)
Call Method my_sub with a1(float), b1(float). Save Result
in c1(float)
Print the Result c and c1

Main Returns 0. Main Ends.
```

Fig 6(b) Source Code Documentation for Entire Program

System Gen. Summary Vs. References	Rouge-1			Rouge-L			Rouge-W		
	P	R	F1	P	R	F1	P	R	F1
Method – Expert generated summary	87.48	71.25	79.37	81.59	68.84	75.22	61.24	30.34	45.79
Program – Expert generated summary	50.93	82.5	66.72	44.57	64.16	54.37	33.46	32.54	33
Algorithm	74.18	74.79	74.49	77.73	78.27	78	49.63	38.12	43.88

Fig 7. Average ROUGE Results of the Proposes System

Two types of documentation are generated by the system based on user choice; method-based documentation is given in Figure 6(a) and entire source code documentation is in Figure 6(b). Method based documentation gives the code summary only for user-defined methods of the program. Whereas, detailed documentation can describe all the statements of the program that include user-defined and pre-defined methods. Nearly, 20 programs are tested in this proposed system. All the results are in precise quality. The summary generated by the proposed system is compared with the documentation developed by the software experts. The ROUGE measurement [9] is used to evaluate the quality of the system generated source code documentation. Figure 7 shows the ROUGE metrics (P-Precision, R-Recall, F1-Average) of the automatic source code documentation system under its method-based and program-based documentation. Also, system generated documentation is compared with an expert's generated algorithm since its format looks like an algorithm. Figure 7 shows that ROUGE-1 results are better than ROUGE-L and ROUGE-W. Recall of detailed (program) documentation is better than its Precision in ROUGE-1. Whereas, method-based documentation has better Precision than its Recall under the same category. F1 metric solves this confusion. It shows the proposed system performance is better in method-based documentation than entire program documentation. When compare the system generated detailed-documentation with expert's algorithm, Precision and Recall are very similar. Though the documentation generated by the system is enough to satisfy the end-user, the ROUGE metric results are failed to justify that point. It shows the importance to identify a new evaluation metric to analyze the performances of the source code summarization system without any ambiguity.

5. Conclusion

The proposed system of this work can automatically generate source code documentation for C programs. Existing software documentation tools can generate documentation only for methods of the software code. This restriction is solved by this proposed system. The automatic source code documentation system uses the code summarization technique of NLP along with NLG. This system can document each and every line of the C program. Two major documentation types are offered by this system; method-based documentation and program-wise documentation. The core component of code summarization system, Software Word Usage Model (SWUM) is successfully built using the programming language's syntax-based Context-Free Grammar rules. This automatic source code documentation system reduces the cost of the documentation process of the software life cycle model. Also, this automation can reduce the overall cost of software project development. In this research work, we proposed a system that works only with C-Lite compiler-based C programs. C-Lite is a lightweight compiler and we are unable to check the complete capability of this proposed system prototype against the complicated syntax-based programming languages. In future work, this system will be enhanced and modified in order to generate code documentation for other high-end, objected-oriented programming languages like Java, Python and Web designing programming languages.

References

[1] P. W. McBurney and C. McMillan, "Automatic Source Code Summarization of Context for Java Methods," in IEEE Transactions on Software Engineering, vol. 42, no. 2, pp. 103-119, 1 Feb. 2016.

[2] S. Badihi and A. Heydarnoori, "CrowdSummarizer: Automated Generation of Code Summaries for Java Programs through Crowdsourcing," in IEEE Software, vol. 34, no. 2, pp. 71-80, Mar.-Apr. 2017.

[3] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K. Vijay-Shanker. 2010. Towards automatically generating summary comments for Java methods. In Proceedings of the IEEE/ACM international conference on Automated software engineering (ASE '10). ACM, New York, NY, USA, 43-52.

[4] Pollock L., Vijay-Shanker K., Hill E., Sridhara G., Shepherd D. (2013) Natural Language-Based Software Analyses and Tools for Software Maintenance. In: De Lucia A., Ferrucci F. (eds) Software Engineering. Lecture Notes in Computer Science, vol 7171.

[5] J. Fowkes, P. Chanthirasegaran, R. Ranca, M. Allamanis, M. Lapata and C. Sutton, "Autofolding for Source Code Summarization," in IEEE Transactions on Software Engineering, vol. 43, no. 12, pp. 1095-1109, 1 Dec. 2017.

[6] Francu, Jan and Petr Hnetynka. "Automated Code Generation from System Requirements in Natural Language." e-Informatica 3,2009: 72-88.

[7] E. Hill, L. Pollock and K. Vijay-Shanker, "Automatically capturing source code context of NL-queries for software maintenance and reuse," 2009 IEEE 31st International Conference on Software Engineering, Vancouver, BC, 2009, pp. 232-242.

[8] Lei, Tao et al. "From Natural Language Specifications to Program Input Parsers." ACL (2013).

[9] C.-Y. Lin. "ROUGE: A package for automatic evaluation of summaries." Proceedings of the ACL-04 Workshop, 2004.