The Nucleus ***53****, No. 1 (2016) 56-63*

www.thenucleuspak.org.pk

The Nucleus

ISSN 0029-5698 (Print)

ISSN 2306-6539 (Online)

C++ BUG CUB: Logical Bug Detection for C++ Code

A. Raana[1], [*]M. A. Azam[1], M. A. Ghazanfar[1], A. Javed[1], Y. Amin[1] and U. Naeem[2]

[1]*Faculty of Telecom and Information Engineering, University of Engineering & Technology Taxila, Pakistan*

[2]*School of Architecture, Computing and Engineering, University of East London, London, U.K*

ayesha.raana@yahoo.com; awais.azam@uettaxila.edu.pk; mustansar.ali@uettaxila.edu.pk; ali.javed@uettaxila.edu.pk; yasar.amin@uettaxila.edu.pk; u.naeem@uel.ac.uk

ARTICLE INFO

Article history :
Received : 25 August 2015
Revised : 29 February, 2016
Accepted : 08 March, 2016

Keywords :
Software quality,
Logical bugs,
Tokenization,
Classification,
Decision tree,
Dependency,
Function

ABSTRACT

Quality is seen as one of the key aspects for efficient and robust Software development. One of the ways to ensure quality is to ensure that developed software systems' code istotally free of syntax, real time and logical bugs. Despite careful development process, there is always room for these bugs to stay in developed system. Many of the syntax and logical bugs escape from detection in testing phase, which has great impact on the quality and reliability of system and business value as well. These are usually Logical Bugs, which can be difficult to find and which can lead to frustration for the development team. To alleviate the overhead of static analysis of code performed by the developer to detect logical bugs, a system is proposed to detect these bugs in C++ code. The system has been engineered using tokenization concept -prerequisite for bug detection- followed by rule-based algorithm that is designed for logical bugs' detection. A decision tree based approach has also been applied in order to classify the detected bugs. The system is also able to extract dependency among all the methods/functions written in the input code. Both tasks; bug(s) detection and function dependency are performed in one pass which makes the system efficient.

1. Introduction

Software systems are the backbone of every field in this modern age of technology, and so reliability of software systems is becoming ever important. Unfortunately, all types of software bugs either in design, code or integration of software units, continues to be sporadic and account for the key causes of system failure/crash. Most important bugs in code reduce software quality, increase the maintenance cost and become the cause of development schedule suspension. Department of Commerce National Institute of Standards and Technology of US conducted a study and concluded that software code bugs are so ubiquitous and harmful that they annually cost $59 billion that is about 0.6% of the gross domestic product [1]. Detection and fixing of the bugs is really time consuming and one of the most difficult tasks in Software Development Life Cycle (SDLC) [2].

Software defects/bugs are actually programming errors or failures in computer program or system that causes unexpected results or behaviors. A software system may suffer from many types of bugs, depending upon how the developer counts. Most common types are: Compilation/Syntax, Execution/Run-time and Logical errors. Novice developers are usually found stressed about syntax and logical bugs appearing in their written codes. Time spent on correcting these bugs may dishearten them with programming. Almost every compiler generates error messages about syntax violation and memory exceptions but excessive logical errors are least likely to be caught during the compilation phase because this phase mainly focuses on specification confirmation. Currently to detect logical bugs, static code review [3] is expected which is off-line task conducted by the code reviewers without compiling or executing the code. During code review activity, reviewer reads the code line by line, understands its behavior as well as structure and then be able to detect the logical and real time bugs after a tiresome manual review analysis.

With aim of minimizing the logical bug rate, there is a need of such system that will efficiently catch these bugs and then classify them. In order to relieve the developer's headache of manually detecting and then correcting those bugs that are beyond the compiler's scope, a system is proposed to detect most commonly occurring logical bugs in C++ codes. These are: uninitialized variables, extra semicolon in front of conditional loops, missing 'break' key word in switch statement cases and use of assignment operator in if-conditional statement to check the equality [4-6]. The proposed system also has the feature to extract dependency among all the methods/functions to understand structure of the input code before making any

[*] Corresponding author

change in the code for correcting bugs. Work done in this paper, subdues novice developers from the detection of above mentioned logical bugs.

This paper comprises of six sections. Section II provides critical review of the related work done in the field of defect detection of software system and dependency among all the methods/functions in the C++ code. Section III, comprises on the detailed description of the Logical Bugs to be detected. Section IV, explains the methodology and Section V describe the results of the proposed work in the form of percentage accuracy. In Section VI closing annotations are given sustaining future research directions.

2. Related Work

A plethora of research has been done in the field of Software defect detection especially for syntax errors. A lot of approaches and tools have been developed by using the various Data Mining techniques.

Kartha et al. [7] describe that the process of collecting data from different perspectives, examining it and then summarizing that data into useful and meaningful information is called Data Mining. This data mining process makes it possible to understand the relationship between gathered data and disclose hidden patterns and trends. Data mining techniques are mature enough that by applying these on historical data of defects, most commonly occurring causes of defects can be identified. The commonly used data mining techniques to predict defects and for untailored analysis are Classification, Clustering [23] and Association Mining [8]. Kartha et al. proposed a tool for automatic defect cause analysis of software defects and for this purpose historical data is used as an input. The tool comprises three stages, 1. Hadoop File System [9] and Map reduce [10] are used to store and process the defect dataset. 2. Defects Classification 3. Fault Tree Analysis. Decision Tree algorithms are used for last two stages. Also the tool has two segments, one is Training and the other is Operational. In Training segment the root cause for each defect is known and a model is formed for unseen data whereas in Operational segment unseen defects are fed to the system and the system performed all its processing as per learning from the Training phase. These defects are visible to the analyst as dashboard and end result of this phase is root cause report.ID3 (Iterative Dichotomiser 3) decision tree algorithm [11] is used to classify the defects and to generate the fault tree for the identification of root causes.

Owens et al. [12] in their research work presented the survey analysis of using the tool 'PURIFY' to find out the reasons of memory access errors in released software systems. Five types of memory errors are considered and classified, these are: Uninitialized memory read, Array bounds read, Array bounds write, free memory read and Free memory write error. These errors are intentionally excluded from the Operating system library. The 1st combination of software and hardware system on which PURIFY is available is Sun SPARC workstations running the UNIX operating system version SunOS 4.1.3. Only C and C++ programs are scrutinized and memory access type errors along with their relevant libraries are taken into account in each software package. The tool proposed by Owens et al. [12] deals only with memory errors and doesn't support other logical errors detection.

Dommati et al. [13] worked in their paper on analysis and classification of network bugs using Naïve Bayes approach [14]. They achieved feature extraction and feature selection based upon the static analysis of bug reports after performing noise reduction in data. With the help of bug reports, feature extraction is done according to the operating system, product related bugs and different networking protocols (BGP, IPv4, IPv6, TCP) to which the reports fit in. 'Information Gain' criteria are used to rank different bug specific features. They proved that only 'Word Information' is not enough to rely upon for multiclass classification, rather semantics of the bug information is necessary. Two event models namely 'Bernoulli' and 'Multinomial' are considered well for classifying the extracted features. Also by the use of these event models better accuracy can be achieved on the basis of extracted features as compared to the use of only Word Information for classification.

Naidu et al. [15] proposed an approach for defect detection using Decision Trees. They used ID3 [11] to classify defects after identification. The following attributes of data were used to classify the defects: 'volume', 'program length', 'difficulty', 'effort' and 'time estimator'. The identified classes were: Blocker Type, Critical Type, Major Type, Minor Type and Trivial Type. A pattern mining approach such that Association Rule mining was used to identify the defect patterns in the data set file. Finally they employed Quality Matrices (Defect Density and Defect Removal Efficiency) to assure the quality of their proposed system. Their proposed system proved useful in finding out the severity of the detected defects.

Liu et al. [16] made a comparative study between CBA2 and other Rule Based Classification methods to ensure that either Classification algorithms based on Association rules are appropriate for predicting software faults or not. They also investigated either Rule Set learned by one data set of defects was valid for other data sets or not. The result of the experiments conducted on data sets was compared with other classification algorithms and confirmed agreeable performance without any loss of comprehensibility.

Wang et al. [17] proposed a new tool to extract symbol level and subsequently module level dependency from the large C/C++ written programs. The system first finds symbol level dependency using LLVM [18] and then these separate pieces of symbol-level dependency are connected to get the module-level dependency. The system's performance is evaluated on the Chromium Project containing about 6 MLOC.

In view of the above mentioned work, it is obvious that there is no clear evidence that one technique for bug detection is better than the others when used independently. The experiments in the above mentioned related work were conducted in different contexts: different programs were examined, detection of different types of bugs was done, and different techniques were used in data mining domain for bug detection. As the literature reveals, the research done for the detection of logical bugs is very limited so there is still room to work for the development of such system that will efficiently detect the logical bugs in source code of any programming language.

3. Logical Bug Types

Logical bugs are not detectable during compilation of the code so the program compiles and runs successfully. However, it won't generate the expected results. The proposed system is efficiently able to detect the most commonly occurring logical bugs mentioned below [24].

3.1 Uninitialized Variable

A variable is read before its actual value written to the memory. In C++ codes such variables contain dummy values except zero and at the end cause the program to produce unexpected results.

```
int count;
while(count<=10)
{
Cout<< "value= " count<<endl;
Count++;
}
```

Fig. 1: Uninitialized variable

```
intx,y;
int sum = x+y;
cout<<"Enter two numbers to sum";
cin>>x>>y;
cout<<"The Sum is: " <<sum;
```

Fig. 2: Uninitialized variable

Considering the code snippet in Fig. 1 the variable '*count*' may contain any value within the range of 'int' data type, and in that case the condition of loop can never be true. In Fig. 2 the variable 'sum' will not actually contain the sum of the numbers entered by the user because in C++ the assignment is one time deal. In example code snippet, '*x*' and '*y*' are not initialized before assigning to the variable '*sum*' so the result will be any random value without considering the user inputs.

3.2 Extra Semicolon

In C++ codes, semicolon doesn't go after conditional loops except the do-while. The program will function abnormally if you put semicolon in front of either 'for' or 'while' loop. The code written in Fig. 3 will just output '10' rather than the sequence from 0 to 9.

```
int a;
for(a=0; a<10;a++)
{
Cout<<"Value=" a<<endl;
}
```

Fig. 3: Extra semi colon after loop condition

3.3 Missing 'break' keyword in Switch Statement

In C++ programs, the cursor does not jump out of the scope of switch statement until the '*break*' keyword is found; otherwise all the cases will be executed one after the other from where the match found. The output generated by the code written in Fig. 4 is 'One' and 'Two' even though only 'One' is expected as the final output.

```
intcheck = 1;
switch(check)
{case 1:
cout<<"One"<<endl;
case 2:
cout<<"Two"<<endl;
break;}
```

Fig. 4: Missing '*break*' in switch structure

3.4 Assignment Operator in 'if statement'

In computer languages, single equal sign always indicates Assignment, Not Equality. If a single equal sign is placed to check the equality, the program will assign the value from right side to the identifier on the left side and the result of such statement will be true.

```
inta,b=1, c=0;
cout<<"Enter the Value for a";
cin>>a;
if (a=5)
{c=a+b}
```

Fig. 5: Assignment operator in if statement

The output of the code written in Fig. 5 will always be '6', because on every execution of the if-statement the variable 'a' is assigned the value 5 rather than checking whether the value of the variable 'a' entered by the user is 5 or not.

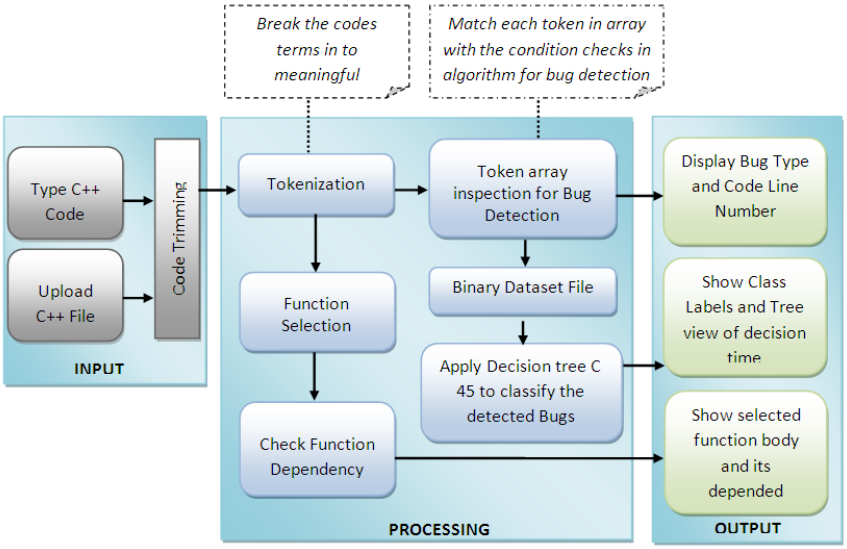

Fig. 6: Proposed compiler

In the case of above mentioned 'Non-Errors' errors, the compiler doesn't detect any problem and no error message will be produced that is such errors are not errors as far as compiler is concerned. After getting the output, entire burden is on the programmer to detect what is wrong with the code. The best way to handle such errors is to avoid them in the first place. In this research work, a software framework is proposed which handles logical errors automatically just by taking the C++ code as input. The developer need not to go through the code line by line to detect the exact statement suffering of logical bug.

4. Proposed Compiler

Fig. 6 shows the structure of proposed compiler. It works in three steps: 1) Input C++ Code, 2) Processing (Bug Detection and Function Dependency), and 3) Output.

i. Input component is actually user interface where an untested C++ code file is fed to the system to check any possible logical bugs mentioned above;
ii. Processing unit is core of the system and comprises two sub-parts, Logical Bug Detection and Function Dependency.

Bug detection further consists of three main steps: a) Tokenization, b) Algorithm to match Buggy Token c) Classification.

4.1 Tokenization

M. Johnson and J. Zelenski in their proposed work [19] explained lexical analysis as a process where the stream of characters making up the source program is read from left-to-right and grouped into tokens. Tokens are sequences of characters with a collective meaning [19] and Tokenization of the text input file of C++ code is prerequisite to bug detection. Here Tokenization process breaks the stream of input code text into words, phrases, symbols or other meaningful rudiments called Tokens. This process ignores the comments, new line space, tab space and also trim the input code. The Token's list then becomes input for further processing such as parsing or text data mining [20]. Common concept of Tokenization is to separate tokens by doing analysis based upon white space characters, line breaks or on the basis of any other delimiter such as some punctuation marks. This concept is applicable for doing tokenization of plain text written in English language where there is a space after each word but same logic is inapplicable for high level languages like C++ because there is no such defined pattern of spaces, new line characters or punctuation marks [25]. While considering a semantically and syntactically correct code we were unable to rely on spaces or anything like that for tokenization. To overcome this bottleneck Algorithm-1 is proposed, using this algorithm the key which always to be looked for in a code for tokenization is the next expected token which has the probability to occur after getting one token.

4.2 Algorithm 1: Tokenization

Input: C++ Code

Output: T= {t1, t2, t3. . .tn}, Tokens of input code

1: Trim the Input code
2: Break the Trimmed code into character stream
3: length of character stream Array
4: repeat
5: Let C: C= {c1, c2, c3 . . .ck} take characters one by one
6: concatenate characters
7: until all characters convert into meaningful tokens
8: return {t1, t2, t3, . . ., tn}

4.2 Algorithm to Match Buggy Token

To detect the logical bugs a very simple algorithm is designed which work somewhat like other string matching algorithms. After successful tokenization process the array of tokens is passed to the bug detection algorithm that scans the token array from start to end aiming to get exact match of the logical bugs' tokens with the written pattern string checks in the proposed system.

If match found, the system highlights the type of bug detected and displays the line number in the source code containing that bug on the user dashboard. The designed algorithm minimizes the total number of comparisons between tokens array and bug patterns defined in the algorithm by continuing from the next token in case of any faulty token found. While checking for logical bugs, binary data set file is being updated for each input code either match found or not.

4.3 Algorithm 2: Matching Algorithm

Input: T= {t1, t2, t3. . .tn}, Tokens

Output: Highlighting the line number of Buggy token

1: calculate length of array containing tokens *T*

2: repeat

3: Get tokens one by one from the array *T* and check either bug condition true or not

4: until *n* tokens are checked

5: return Line number of buggy token or No bug found

4.4 Classification

Classification technique [4] assigns items in a set to target classes. After one time identification of classes, the system deduces the rules to preside over these classes. Classification approaches are fed with the training set where all data and their associated classes are pre known and the Classification algorithm build model on the basis of its learning, which then can be used for unseen data. In software engineering, classification algorithms have been used to engineer software prediction models which are used to predict software defects and for such other purposes. Here after full execution of bug detection algorithm, the dataset file is fed to WEKA and decision tree C4.5(J48 in WEKA) is selected to classify the detected bugs. The classes build based upon the type of bugs to be detected. The Data Set used here is Binary and artificially created in parallel to the execution of Bug Detection code. As the dataset is not too large, C4.5 algorithm is quite suitable to be used for the classification [26]. Classification part has two folds:

- Training Fold: in which classes are pre labeled and the decision tree is trained according to this known data set.
- Operational Fold: in this fold any unseen data of logical bug(s) is entered to predict the class of the detected bug on the basis of learning from Training Fold.

Function dependency is to extract dependency among all the methods/functions written in the input code. Dependency can be explained as, suppose there are two methods 'X' and 'Y' in the input code, if any change in the method 'X' forces to modify the method 'Y' accordingly to maintain the code's structure and to keep it in exact functionality as well then there is dependency from Y to X.

Before making any change in the code, first key step is to endow the user with precise flow of the code. Dependency extraction makes the structure of the input code understandable and will help the user while correcting the bugs detected by first part of the proposed system. To check dependency level of the inputted code, the system first scans for all the functions used and then finds dependent functions based upon the definition of each function prior enlisted.

Algorithm 3: Function Dependency: Find Function Definition in the complete code

Input: T= {t1, t2, t3. . .tn}, Array of Tokens

Output: Function name, Start point and End point

1: repeat

2: find parenthesis '()' in the array *T*

3: if '()' NOT∈ keywords {if, else if, for, while, do}

THEN

Loop through Left Tokenized Array

IF semicolon

THEN

BREAK

ELSEIF Function opening Braces

THEN

SAVE Function Name is COMBOBOX Item

SAVE Function Starting point

LOOP

Repeat step 3

OR

IF Function Closing Braces

THEN

SAVE Function Ending point

NEXT

NEXT

NEXT

END

Algorithm 4: Function Find DEPENDCE Function CALL

Input: Function name, Start point and End point

Output: Function Body and Dependent Functions

GET Selected Function NAME in COMBOBOX

GET starting and ending points of the Selected Function

LOOP through Tokenized Array from Function Starting to ending point

ADD in codesnap

Find parenthesis in the selected Token array

IF parenthesis NOT belong to if, else if, switch, for, while, do

THEN

ADD identifier name in ftndep

NEXT

DISPLAY codesnap

IF ftndep NOT Empty

THEN

DISPLAY codesnap

ELSE

DISPLAY "No Function is depending"

END

Output is the dashboard for the user on which three things are displayed; one is the window containing the bug type along with the line number of the source code suffering of that detected bug, second is function dependency view showing the body of every function and all its dependent function(s) and third is the tree view of detected bug's classes.

5. Results and Discussion

Proposed system is evaluated on the basis of two static code metrics; LOC (Line of Codes) and Source Code Complexity measure by total number of functions defined in the input code.

The system is actually proposed for novice developers who mostly start their software development life in their university time period. Here to evaluate the proposed system, C++ programs written by engineering university students (2nd year) are executed to get results of the system. Multiple C++ program writing tasks were assigned to the students. They were asked to produce the predefined output of the given programming tasks also to keep record of the correction attempts. The codes are of different lengths and different from each other but following same guideline and format for example there must not be spaces at the end of code. It was observed that 80% of the students had to manually review their written code more than one time to correct the logical bugs to get the required output. The proposed system reduces this tiresome manual review activity for bug detection. In 2nd phase of program writing students were asked to intentionally add some logical bugs and to keep the record of these bugs. Each program file is fed to the system to calculate system accuracy for bug detection considering LOC as parameter.

5.1 *Experimental Results*

The following general equation is used to calculate Accuracy of the system :

Accuracy% = (Number of bugs detected / Total number of Bugs fed to the system) * 100

Table 1: System Accuracy with respect to lines of codes (LOC)

Parameter	LOC	Actual Bugs	Total Bugs	Detected Bugs	Total Detected Bugs	True Positive %
Program File < = 100 LOC	94	3	9	2	7	77%
	74	2		2		
	46	3		2		
	69	1		1		
Program File > 100 LOC	172	3	22	1	15	68%
	205	6		5		
	143	5		4		
	658	8		5		

Every existing C++ editor must contain compiler for syntax error detection and the detection of Syntax errors is considered standard. Due to this standard consideration the proposed logical bug detector's accuracy rate is compared with the detector of syntax errors. Neelima et al. [21] proposed a system shown in Fig 7 for the detection of syntax errors in C language codes. Here the proposed system (C++ BUG CUB) is compared with Neelima's system. The feature Accuracy of both systems is evaluated taking total bugs into consideration. Table 2 and
Fig 8 below, reveals that the proposed system for logical bug detection is almost reaching the accuracy level for the detection of syntax bugs.

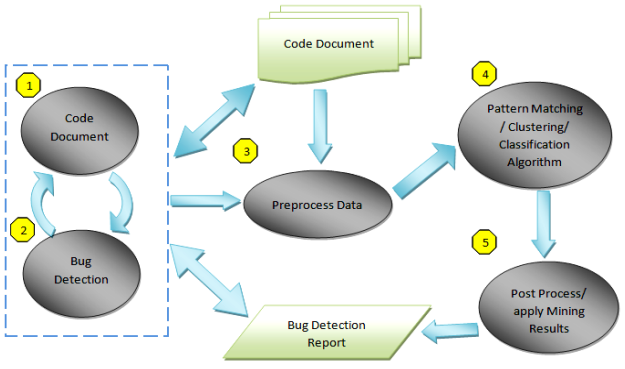

Fig. 7: Neelima's proposed system

To calculate complexity of any input code, Sequential Cyclomatic Complexity (SCC) [22] which considers chain of function calls is used in this research work and shown in Equation 2.

Table 2: C++ BUG CUB vs Neelima's system

System	Number of Bugs	Accuracy %
C++ BUG CUB	Bugs<=15	77%
	Bugs>15	68%
Neelima's	Bugs<=15	70%
	Bugs>15	75%

$$SCC(fn)=CC(fn_{cg})+\sum_{i=1}^{n} CC(fn(i)) \quad (2)$$

(Here fn_{cg} is calling function, and n is number of function calls.)

Ultimately Sequential CC of any program is equal to sum of SCC of all the functions of the program as shown in Equation 3.

$$SCC(program)=\sum_{i=1}^{m} CC(fn(i)) \quad (3)$$

(Here 'm' is the number of function.)

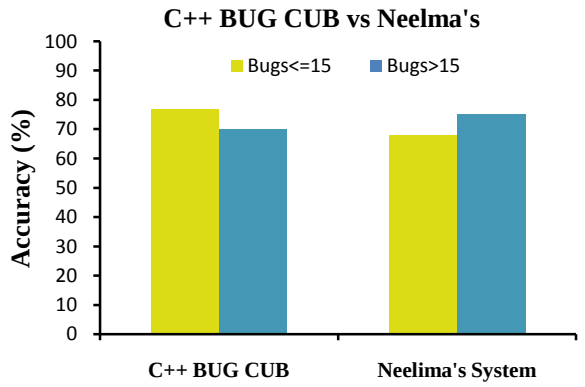

Fig. 8: C++ BUG CUB vs Neelima's system

Table 3 shows that proposed system is almost taking same time to execute the input code of lines below 1500. SCC depends upon total number of calling and called functions, as complexity of the input code increases execution time may increase in seconds but not too much. Also the calculated time values vary depending upon the structure of the input code that is indenting of the code. This is because input code is trimmed and white spaces and comments are ignored before the execution of rule-based bug detection algorithm.

7. Conclusion and Future Work

In this paper, a system is proposed to lessen the burden for developers to detect the most commonly occurring logical bugs in C++ code, which escape in testing phase and ultimately producing the output not according to the requirement. Currently the proposed system is able to detect four types of logical bugs (Uninitialized Variables, Extra Semicolon go after conditional loops, Assignment operator in if-conditional statement to check equality, and missing break keyword in switch statement cases). The system is engineered using the concept of Lexical Analysis and Tokenization, algorithm for actually detecting bugs and to extract function dependency of input C++ code is proposed and in the end decision tree is used to classify the detected bugs. A data set file is kept updated after the execution of any C++ code file either containing logical bug or not.

Table 3: Sequential cyclomatic complexity (SCC) and execution time of the System with respect to LOC

Parameter	LOC	NOF	Dependent Function	SCC	Execution Time(Sec)
Program File<=100 LOC	94	2	1 on main	1	10
	74	2	1 on main	1	9
	46	2	1 on main	1	10
	69	3	2 on main	2	10
	172	2	1 on main	2	10
Program File>100 LOC	205	5	4 on main	4	9
	143	2	1 on main	1	9
	658	7	6 on main	5	10

The sample C++ codes used as input to the developed system are divided into two categories: 1. codes below 100 lines, 2. codes above 100 lines. Accuracy of the proposed system is evaluated based upon two static code analysis metrics; LOC (Lines of Code) and NOF (Number of Function). Execution time calculation result shows that the proposed system saves much time-cost and effort of manual analysis to detect harmless logical bugs while seeing but destructive in actual. The proposed system is compared with Neelima's system (Syntax Bug Detection in C language code) and accuracy level is almost the same although it is harder to detect logical bugs rather than syntax bugs.

This research work has enough room; the proposed system can be integrated into some other bug tracking tools that have been developed to detect the syntax and real time bugs of the C++ code(s). The scope of this work includes extension to other types of bugs such as Real Time Errors, Memory Leak Errors and implementation for other programming languages as well.

References

[1] Z. Li, "Using data mining techniques to improve software reliability", Avaialble:https://www.ideals.illinois.edu/ handle/ 2142/11273, 2006.

[2] Y. Zhang, Y. Liu, L. Zhang and Y. Shi, "A data mining based method: detecting software defects in source code", 2nd Int. Conf. on Software Engg. and Data Mining (SEDM), Chengdu, China, pp. 607-612, 2010.

[3] H. Uwano, M. Nakamura, A. Monden and K. Matsumoto, "Analyzing individual performance of source code review using reviewers' eye movement", Proc. of Symposium on Eye Tracking Research & Applications ACM Digital Library, pp. 133-140, Mar 27, 2006, ACM.

[4] "Common beginner C++ programming mistakes", Available:http: //fd.valenciacollege.edu/file/grhodes4/CommonBeginnerMistakes1 .pdf. [Accessed: 1-08, 2015].

[5] "Eight C++ programming mistakes the compiler won't catch", Available: http://www.learncpp.com/cpp-programming/eight-c-programming-mistakes-the-compiler-wont-catch/. [Accessed: July 2nd 2007]

[6] "10 Common programming mistakes in C++", Available:http:// alumni.cs.ucr.edu/~nxiao/cs10/errors.htm.

[7] R. Kartha and V. Nair, "Data mining for causal analysis of software defects", Master of Science Thesis, Technische University Eindhoven, Int. J. Comp. Sci. Mobile Comput. (ICMIC), pp. 1-7, December, 2013.

[8] P. J. Kaur and Pallavi, "Data mining techniques for software defect prediction", International Journal of Software and Web Sciences (IJSWS 12-347), vol. 3, no. 1, Feb 2013, pp. 54-57.

[9] K. Shvachko, H. Kuang and S.Radia, "The hadoop Distributed file system" Mass Storage Systems and Technologies (MSST), IEEE 26th Symposium on, 3-7 May 2010, pp. 1-10, Incline Village, NV. Available:IEEE Xplore, http://www.ieee.org.

[10] J. Dean and S.Ghemawat, "MapReduce: simplified data processing on large clusters", Communications of the ACM, vol. 51, no. 1, pp. 107-113, January 2008.

[11] W. Peng, J. Chen and H. Zhou, "An Implementation of ID3— Decision Tree Learning Algorithm", Available: http://cis.k.hosei.ac.jp/~ rhuang/Miccl/AI-2/L10 src/Decision Tree2. pdf 2009.

[12] H. D. Owens, B. E. Womack and M. J. Gonzalez, "software error classification using purify", International Conference on Software Maintenance, CA, USA , Nov. 4-8 1996, pp. 104-113.

[13] S. J. Dommati, R. Agrawal, G.R.M. Reddy and S. Kamath, "Bug classification: feature extraction and comparison of event model using naïve bayes approach", Int. Conf. on Recent Trends in Comp. Inform. Engg (ICRTCIE), Pattaya, April 13-15, 2012.

[14] A. McCallum and K. Nigam. "A comparison of event models for naive bayes text classication", Available:http://citeseerx.ist. psu.edu/viewdoc/download?doi=10.1.1.65.9324&rep=rep1&type= pdf 1998.

[15] M. S. Naidu and N. Geethanjali, "Classification of defects in software using decision tree algorithm", Int. J. Engg. Sci. Tech. (IJEST), vol. 5, no. 6, pp. 1332-1340, June 2013.

[16] B. Liu, Y. Ma and C.K. Wong, "Improving an association rule based classifier", Proc. of the Fourth European Conference on Principles and Practice of Knowledge Discovery in Databases, pp. 504-509, 2000.

[17] P. Wang, J. Yang, L. Tan, R. Kroeger and J. D. Morgenthaler, "Generating precise dependencies for large software", Managing Technical Debt (MTD), 2013 4th International Workshop on, IEEE, 20-20 May, pp. 47-50, 2013, San Francisco, CA. Available: IEEE Xplore, http://www.ieee.org.

[18] C.A. Lattner, "LLVM: An infrastructure for multi-stage optimization", Graduate College of the University of Illinois at Urbana-Champaign, 2002, pp.1-56.Avaialable: http://www.llvm. org/pubs/2002-12-LattnerMSThesis-book.pdf.

[19] M. Johnson and J. Zelenski, "Lexical Analysis", http://dragonbook. stanford.edu/lecture-notes/Stanford-CS143/03-Lexical-Analysis. pdf, June 25, 2008.

[20] MA Hearst, "Text data mining: Issues, techniques, and the relationship to information access", in journal Presentation notes for UW/MS workshop on data mining, pp. 112-117, July 1997.

[21] V. Neelima, Annapurna. N, V. Alekhya and B. M. Vidyavathi, "Bug detection through text data mining", Int. J. Adv. Res. Comput. Sci. Softw. Engg., vol. 3, no. 5, pp. 564-569, May 2013.

[22] KB Kumar, J. Gyani and G. Narsimha, "Sequential cyclomatic complexity over a chain of function calls", Int. Conf. on Networks and Information, PCSIT, vol. 57, 2012, IACSIT Press, Singapore.

[23] P. Devi and R. Ranjan, "enhanced bug detection by data mining techniques", Int. J. Comput. Engg Res. (IJCER), vol. 4, no. 7, pp. 19-27, July 2014.

[24] D. Radosevic and T. Orehovacki, "An analysis of novice compilation behavior using Verificator", Proceedings of the ITI 2011 33rd, Int. Conf. on Information Technology Interfaces, June 27-30, 2011, Cavtat, Croatia. Available: IEEE Xplore, http://www.ieee.org.

[25] S.H Abid, S. Zehra and H. Iftikhar, "using computer aided software for teaching and self learning", Proceedings of the ICL 2011 14th Int. Conf. on Interactive Collaborative Learning, 21-23 Sept. 2011, Piestany. Available: IEEE Xplore, http:// www.ieee.org.

[26] D. Lanvanya and K. U. Rani, "performance evaluation of decision tree classifiers on medical datasets", Int. J. Comp. Appl., vol. 26, no. 4, July 2011.