



International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 4, April 2016

Reduction of Cyclomatic Complexity by use of Mutation Testing

Vijay Kumar Sinha¹, Rubaljeet Kaur²

Research Scholar, IKG PTU, Kapurthala, Punjab, India¹

Assistant Prof, CGC , Landran ,IKGPTU University, Kapurthala, Punjab, India²

ABSTRACT: Reduction of software complexity is a major challenge among the software industries. Softwares use different algorithms to implement their objectives to deliver a commercial product. Although algorithms are pre tested for its complexities at individual levels. However complexities increases manifold during its actual implementation in association with other components to make it functional. This happen due to following reasons:

Most of the software testing mechanisms meant only to test the software for its possible bug, errors and to obtain the desired result / output. Its not used for testing of its complexities that affects its performance in terms of time and space ultimately. Testing strategies and test cases are rarely used for reduction of software complexities. The propose method is very effective in reducing the software complexities arise due to unwise use of programming techniques and mismanagements. Usually this complexity arises due to:

- a) Code redundancy.
- b) Unwise implementation strategies.
- c) Poor programming skills.
- d) Unnecessary declaration statements. Etc.
- e) Overburden of loops.
- f) Unwise selection of algorithms

This increases the overall software complexities.

Our technique successfully applied in reduction of complexities as well as software testing by using the same mutation testing.

In this successfully demonstrated the dual application of Mutation testing .thus by this approach we need not to apply two different methods separately for Software testing as well as complexity testing . Thus this is a significant reduction in time while software testing.

KEYWORDS: Mutation testing, Software complexity, Test cases

I. INTRODUCTION

Software evolution is the software system dynamic behaviour while it is maintained and improved over its lifetime [15]. Software systems usually evolve to x problems, accommodate new features, and improve their quality. In order for the software to survive for a long period, it needs to evolve. The changes that the software undergo are formally categorized as corrective, preventive, adaptive and perfective maintenance.

One of the maintainability characteristics of the ISO/IEC Square quality standard is Modularity [13]. In this standard, modularity is de ned as "a degree to which a given system or a program is composed of discrete components; such that a change in any component has minimal impact on the other components" [13]. Modularity is further decomposed at that standard into two main dimensions, coupling and complexity.

McCabe's Cyclomatic Complexity (CC), first introduced in 1976 [20], measures the software complexity using its flow graph. In practice, CC counts the decision points in the software. The CC metric definitions are independent of the syntax and the productivity differences among programming languages. Complexity measures identify components that have the highest complexity. Most likely, these components contain bugs, because complexity makes them harder to understand.

Complexity measures are very useful in identifying complex program units. Numerous research studies investigated the relationship between program complexity and maintain-ability .cost. Large values for CC hinder the software evolution since they make the program both di cult to change and to understand.



International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 4, April 2016

Developers have difficulty in understanding complex software units, and the complexity might vary between different implementations to solve the same problem [6]. In order for the software to evolve, software has to be updated regularly with added features, which in part add more complexity to the software [6]. However, it is essential to strive for minimizing the complexity to facilitate adding features, or xing bugs in the future [6]. Manduchi and Taliercio [19] investigated the complexity evolution of closed-source systems in which they showed that almost all complexity measures increased with different pace. MacCormack et. al [18] found that restructuring the software design of The remainder of this paper is organized as follows. Section 2 states Lehman's laws of software evolution. Section 3 discusses the previous open-source projects studies. Section 4 states the metrics used in this study. The empirical study is given in Section 5. Some threats to validity are presented in Section 6. Conclusions are presented in Section

II. RELATED WORK

Approaches to Code Complexity Testing-Cyclomatic Complexity

Two questions coming to mind while doing the code complexity testing are:

The collections of various metrics are as under

- 1) McCabe Metrics: are based on Cyclomatic Complexity, $V(G)$.
 - 2) Execution Coverage metrics: are based on any of Branch, Path or Boolean coverage.
 - 3) Code Grammar metrics: are based around line counts and code structure counts such as Nesting.
 - 4) OO metrics: are based on the work of Chidamber and Kemerer.
 - 5) Derived Metrics: are based on abstract concepts such as under stability, maintainability, comprehension and testability.
 - 6) Custom Metrics: are imported from third party software / systems, e.g. defects count.
- McCabe IO provides for about 100 individual metrics at the Method, Procedure, Function, Control and Section / paragraph level. 40 additional matrices are available at the class / file and program level.

Categories of Metrics: There are three categories of metrics

- 1) McCabe Metrics.
- 2) OO Metrics.
- 3) Grammar Metrics.

When collecting metrics, we rely upon subordinates who need to 'buy into' the metrics program. Hence, it is important to only collect what you intend to use. We should remember, 'The Hawthorne Effect' which states that when you collect metrics on people, the people being measured will change their behavior. Either of these practices will destroy the efficiency of any metrics program.

Let us discuss the above mentioned three metrics categories

1) McCabe Metrics:

a) Cyclomatic Complexity, $V(G)$: It is the measure of the amount of logic in a code module of 3rd and 4th generation languages. If $V(G)$ is excessively high then it leads to impenetrable code i.e., a code which is at higher risk due to difficulty in testing. The threshold value is 10. When $V(G) > 10$; then the likelihood of code being unreliable is much higher. It must be remembered that a high $V(G)$ shows a decreased quality in the code resulting in higher defect that become costly to fix.

b) Essential Complexity: It is a measure of the degree to which a code module contains unstructured constructs. If the essential complexity is excessively high, it leads to impenetrable code i.e. a code, which is at higher risk due to difficulty in testing. Furthermore, the higher value will lead to increased cost due to the need to refactor or worse, reengineer the code. The threshold value is 4. When the essential complexity is more than 4 than the likelihood of the code being un-maintainable is much higher. Remember that a high essential complexity indicates increased maintenance costs with decreased code quality.

Some organizations have used the Essential Density metric (EDM) and it is defined as

$EDM = (\text{Essential complexity}) / (\text{Cyclomatic complexity})$

c) Integration Complexity: It is a measure of the interaction between the modules of code within a program. Say, $SO, S1$ are two derivatives of this complexity.

Where SO - Provides an overall measure of size and complexity of a program's design. It will not reflect the internal calculations of each module. It is the sum of all integration complexity in a program.

And $S1 = (SO - \text{Number of methods} + 1)$.



International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 4, April 2016

This is primarily used to determine the number of tests for the 'some test' that is designed to ensure that the application would execute without issues in module interaction.

d) Cyclomatic Density (CD): It is a measure of the amount of logic in the code.

It is expressed as follows

Software Testing

Software Testing is the process of executing a program or system with the intent of finding errors. Or, it involves any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results. Software is not unlike other physical processes where inputs are received and outputs are produced. Where software differs is in the manner in which it fails. Most physical systems fail in a fixed (and reasonably small) set of ways. By contrast, software can fail in many bizarre ways. Detecting all of the different failure modes for software is generally infeasible.

For Verification & Validation (V&V)

Verification and Validation indicated, another important purpose of testing is verification and validation (V&V). Testing can serve as metrics. It is heavily used as a tool in the V&V process. Testers can make claims based on interpretations of the testing results, which either the product works under certain situations, or it does not work. We can also compare the quality among different products under the same specification, based on results from the same test.

For reliability estimation

Software reliability has important relations with many aspects of software, including the structure, and the amount of testing it has been subjected to. Based on an operational profile (an estimate of the relative frequency of use of various inputs to the program, testing can serve as a statistical sampling method to gain failure data for reliability estimation.

Mutation Testing

Mutation testing process starts with the generation of mutants. A mutant is actually a copy of original program containing one permutation (fault) which is syntactically correct. These permutations or faults are introduced using a pre-defined set of permutations called mutation operators. After generating mutants, the next step is to execute all the test cases against all mutants and compare the outputs with the original program's outputs. If a test case produces different output on a mutant then the mutant is said to be killed by the test case otherwise the mutant is said to be alive. A mutant is called equivalent mutant if no test case can distinguish its output from the original program's output. It is impossible to kill an equivalent mutant because it is semantically equivalent to the original program. The ratio of killed and alive mutants is known as mutation score which indicates how effective a given test case set is. Tester can regenerate test cases to kill the remaining alive mutants and to raise the mutation score because a test case set with higher mutation score is considered more effective.

Kinds of Mutation

_ **Value Mutations:** these mutations involve changing the values of constants or parameters (by adding or subtracting values etc), e.g. loop bounds being one out on the start or `_nish` is a very common error.

_ **Decision Mutations:** this involves modifying conditions to reflect potential slips and errors in the coding of conditions in programs, e.g. a typical mutation might be replacing a `>` by a `<` in a comparison

_ **Statement Mutations:** these might involve deleting certain lines to reflect omissions in coding or swapping the order of lines of code. There are other operations, e.g. changing operations in arithmetic expressions. A typical omission might be to omit the increment on some variable in a while loop.

Value Mutation

Here we attempt to change values to reflect errors in reasoning about programs.



International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 4, April 2016

Decision Mutation

Here again we design the mutations to model failures in reasoning about conditions in programs. As before this is a very limited model of programming error really modelling slips in coding rather than a design error.

Cyclomatic Complexity

A cyclomatic complexity is a calculation of the number of independent paths in a program by computing the cyclomatic complexity (McCabe, 1976) of the program flow graph. For programs without goto statements, the value of the cyclomatic complexity is one more than the number of conditions in the program. A simple condition is logical expression without 'and' or 'or' connectors. If the program includes compound conditions, which are logical expressions including 'and' or 'or' connectors, then you count the number of simple conditions in the compound conditions when calculating the cyclomatic complexity.

Therefore, if there are six if-statements and a while loop and all conditional expressions are simple, the cyclomatic complexity is 8. If one conditional expression is a compound expression such as 'if A and B or C', then you count this as three simple conditions. The cyclomatic complexity is therefore 10. The cyclomatic complexity of the binary search algorithm is 4 because there are three simple conditions at lines 5, 7 and 11.

After discovering the number of independent paths through the code by computing the cyclomatic complexity, you next design test cases to execute each of these paths. The minimum number of test cases that you need to test all program paths is equal to the cyclomatic complexity.

III. PROPOSED ALGORITHM

Proposed Mutation Testing for Complexity reduction

Methodology and tools used:

1. We use case study method for verification and validation of result. The statistical analysis used for the result comparison.
2. Testrail : it is online software testing tool
3. Metric Plugin for Eclips

Tools for Cyclomatic Complexity calculation:

Many tools are available for determining the complexity of the application. Some complexity calculation tools are used for specific technologies. Complexity can be found by the number of decision points in a program. The decision points are if, for, for-each, while, do, catch, case statements in a source code. Examples of tools are

- OCLint - Static code analyzer for C and Related Languages
- devMetrics - Analyzing metrics for C# projects
- Reflector Add In - Code metrics for .NET assemblies
- GMetrics - Find metrics in Java related applications
- NDepends - Metrics in Java applications

Here we take the case of binary search algorithm which is implemented in C++ programming language .

We take Binary Search Algorithm for the test case .

The complexity of Binary search is :

Best Case : $O(1)$

Average case : $O(\log n)$

Worstcase : $O(\log 2n)$

Original C++ Code for Binary search Testing :

Mutation Level : 0 :



International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 4, April 2016

```

#include <iostream.h>
#include <conio.h>
int binary_search(int [],int,int);
main()
{
clrscr();
const int array_size=10;
int array[array_size]={0,6,9,12,20,23,29,32,47,79};
cout<<"*****"<<endl;
cout<<"***** Binary Search *****"<<endl;
cout<<"*****"<<endl;
gotoxy(1,24);
cout<<"*****"<<endl;
cout<<"*****";
gotoxy(1,5);
cout<<"\n The contents of the array are : "<<endl;
cout<<"\n Elements : "<<"\t\t Value:"<<endl;
for(int count=0;count<array_size;count++)
{
cout<<"\t"<<" array ["<<count<<"]"<<"\t\t";
cout<<array[count]<<endl;
}

int searching_element=0;
int flag=0;
cout<<"\n Enter the element you want to find = ";
cin>>searching_element;
flag=binary_search(array,array_size,searching_element);
if(flag!=-1)
cout<<"\n The given element is found at the position array["<<flag<<"].";
else
cout<<"\n The given element is not found.";
getch();
return 0;
}

/*****//-----
binary_search() -----
/*****/int binary_search(int
array[],int array_size,int element)
{
int start=0;
int end=array_size-1;
int middle;
int position=-1;
middle=(start+end)/2;
do
{
if(element<array[middle])
end=middle-1;
elseif(element>array[middle])
start=middle+1;
}
}

```



International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 4, April 2016

```
middle=(start+end)/2;
    }
while(start<=end && array[middle]!=element);
if(array[middle]==element)
position=middle;
return position;
```

Cyclomatic complexity :

Total Numbers of Independent paths =21 , Which represents a very complex coding

Mutation level : 1

Reduction of Lines of codes and function points by using Mutation testing (Using array)

```
#include <cstdlib>
#include <iostream>
using namespace std;

int binary_search(int array[],int first,int last, int value);

int main() {
int list[10];
for (int k=0; k<11; k++)
list[k]=2*k+1;
cout<< "binary search results: "<<binary_search(list,1,21,11)<<endl;
return 0;
} //end of main
int binary_search(int array[],int first,int last, int search_key)
{
int index;
if (first > last)
index = -1;
else
{
int mid = (first + last)/2;
if (search_key == array[mid])
index = mid;
else
if (search_key< array[mid])
index = binary_search(array,first, mid-1, search_key);
else
index = binary_search(array, mid+1, last, search_key);
} // end if
return index;
} // end binarySearch
```

Cyclomatic Complexity of level 1 : 12 which is a Complex coding

Mutation Level 2 : ,/

```
int binarySearch(int arr[], int value, int min, int max){
int pos = -1;
while (max >= min && pos == -1) {
int mid = (max+min)/2;
if(arr[mid] == value){
pos = mid;
```



International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 4, April 2016

```
}else if(arr[mid] < value){  
min = mid +1;  
}  
}else if(arr[mid] > value){  
max = mid -1;  
}  
}  
}  
return pos;  
}
```

Cyclomatic Complexity of level 2 : 6 , It is a structured and well written code.

IV. RESULT AND DISCUSSION

In the 0 –level the cyclomatic complexity comes 21 which is a very complex coding . It reduces in the level 1 for the same algorithm and which is 12 .This is a simpler than level 0 but still a complex code . It further minimized in the level 2 at the level of 6 which is a coding of well written and structured code . This made possible by means of mutation testing which is able to change the code and the programming patterns.

V. FUTURE SCOPE

This paper explores a new approach to test both validity of software as well as control over software complexity to serve a more quality software product to face the competitive market for its excellence. This approach can be extended to validate design of software. The mutation testing is also very useful while testing the compatibility of software for a specific OS platform.

REFERENCES

- [1] S. Androutsellis-Theotokis, D. Spinellis, M. Kechagia, G. Gousios, S. Evdokimov, B. Fabian, O. Gunther, L. Ivantysynova, H. Ziekow and M. A Lapre, "Open source software: A survey from 10,000 feet. *Foundations and Trends in Technology, Information and Operations Management*, vol. 4, nos. 3-4, (2011), pp. 187-347.
- [2] E. J. Barry, C. F. Kemerer, and S. A Slaughter, "How software process automation affects software evolution: a longitudinal empirical analysis", *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 19, no. 1, (2007), pp. 1-31.
- [3] S. Bouktif, G. Antoniol, E. Merlo, and M. Neteler, "A feedback based quality assessment to support open source software evolution: the grass case study", In *22nd IEEE International Conference on Software Maintenance*, 2006. ICSM'06., pages 155-165. IEEE, (2006).
- [4] A. Capiluppi and J. F. Ramil, "Studying the evolution of open source systems at different levels of granularity: Two case studies", In *Proceedings. 7th International Workshop on Principles of Software Evolution*, 2004., pages 113-118. IEEE, (2004).
- [5] S. Christley and G. Madey, "Analysis of activity in the open source software development community", In *40th Annual Hawaii International Conference on System Sciences*, 2007. HICSS 2007, pages 166b-166b. IEEE, (2007).
- [6] D. P. Darcy, S. L. Daniel, and K. J. Stewart, "Exploring complexity in open source software: Evolutionary patterns, antecedents, and outcomes. In *43rd Hawaii International Conference on System Sciences (HICSS)*, 2010, IEEE, pp. 1-11, (2010).
- [7] D. P. Darcy and C. F. Kemerer, "OO metrics in practice", *IEEE Software*, vol. 22, no. 6, (2005), pp. 17-19.
- [8] D. P. Darcy, C. F. Kemerer, S. A. Slaughter, and J. E. Tomayko, "The structural complexity of software an experimental test", *IEEE Transactions on Software Engineering*, vol. 31, no. 11, (2005), pp. 982-995.
- [9] T. Dinh-Trong and J. M. Bieman, "Open source software development: a case study of freesbd", In *Proceedings. 10th International Symposium on Software Metrics*, IEEE, (2004), pp. 96-105.
- [10] H. J. C. Ellis, R. A. Morelli, T. R. de Lanerolle, J. Damon, and J. Raye, "Can humanitarian open-source software development draw new students to cs?", In *ACM SIGCSE Bulletin*, vol. 39, ACM, (2007), pp. 551-555.
- International Journal of Hybrid Information Technology Vol.8, No.2 (2015)
Copyright © 2015 SERSC 265
- [11] M. W. Godfrey and D. M. German, "On the evolution of lehman's laws", *Journal of Software: Evolution and Process*, (2013).
- [12] V. K. Gurbani, A. Garvert, and J. D. Herbsleb, "A case study of open source tools and practices in a commercial setting", In *ACM SIGSOFT Software Engineering Notes*, ACM, vol. 30, (2005), pp. 1-6.
- [13] ISO/IEC. Systems and software engineering - systems and software quality requirements and evaluation (square). ISO/IEC 25010 - System and software quality models, (2011).
- [14] D. Landman, A. Serebrenik, and J. Vinju. 'Empirical analysis of the relationship between cc and sloc in a large corpus of java methods', In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, (2014), pp. 221-230.
- [15] M. M. Lehman, "On understanding laws, evolution, and conservation in the large-program life cycle", *Journal of Systems and Software*, vol. 1, (1980), pp. 213-221.
- [16] P. L. Li, J. Herbsleb, and M. Shaw, "Finding predictors of field defects for open source software systems in commonly available data sources: A case study of opensbd", In *11th IEEE International Symposium Software Metrics*, IEEE, (2005), pp. 10.





International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 4, April 2016

- [17] R. Lincke, J. Lundberg, and W. Löwe, "Comparing software metrics tools", In *Proceedings of the 2008 international symposium on Software testing and analysis*, ACM, (2008), pp. 131-142.
- [18] A. MacCormack, J. Rusnak, and C. Y Baldwin, "Exploring the structure of complex software designs: An empirical study of open source and proprietary code", *Management Science*, vol. 52, no. 7, (2006), pp. 1015-1030.
- [19] G. Manduchi and C. Taliercio, "Measuring software evolution at a nuclear fusion experiment site: a test case for the applicability of oo and reuse metrics in software characterization", *Information and Software Technology*, vol. 44, no. 10, (2002), pp. 593-600.
- [20] T. J McCabe, "A complexity measure", *IEEE Transactions on Software Engineering*, vol. 4, (1976), pp. 308-320.
- [21] A. Mockus, R. T Fielding, and J. Herbsleb, "A case study of open source software development: the apache server", In *Proceedings of the 22nd International Conference on Software engineering*, ACM, (2000), pp. 263-272.

BIOGRAPHY

	Vijay Kumar Sinha received his M.Tech. degree in Computer Science and Engineering from Punjabi University , Patiala. Currently he is persuing his Ph.d. degree in Computer Science & Engineering from Punjab Technical University , Kapurthala (Punjab) India.Her research interests include Image Processing , Software engineering , Software components, Software Reuse, Software architecture and software metrics.
	Ms. Rubaljeet received his M.Tech. degree in Computer Science and Engineering from IKG Punjab Technical University, Kapurthala (India) Her research interests include Image Processing, Software engineering , Data base.