ISSN: 0975-766X
CODEN: IJPTFI

Available Online through ***Research Article***
www.ijptonline.com

A STUDY ON COMPLEXITY IN PROGRAMS

Shalu Achamma Sam, M.Thanusha, Dr.R. Manjula
First Year, MTech-CSE, School of Computer Science and Engineering,
VIT University, Vellore, Tamil Nadu-632014.
First Year, MTech-CSE, School of Computer Science and Engineering
VIT University, Vellore - Tamil Nadu-632014.
Associate Professor, School of Computer Science and Engineering
VIT University, Vellore - Tamil Nadu-632014.
Email: shaluasam@gmail.com

Received on: 15.10.2016 *Accepted on: 12.11.2016*

Abstract

The problem of complexity arises where simplicity principles fail. To reduce complexity we use software metrics such as cyclomatic complexity (McCabe) and Halstead complexity measures. In this paper, we bring a comparative study of existing complexity measures on different programs and how to keep a code less complex.

Keywords: Cyclomatic complexity, McCabe's cyclomatic complexity number, Halstead's metrics.

Introduction

First of all, what is complexity in a program? In programming, complexity is when a simple program which we thought end up giving unexpected results. The various reasons behind the complexity of programs are fault tolerance, different/varying modes of system operation and interactions. Fault tolerance increases the complexity of systems in such a way that it demands operation in different sub contexts within the overall context of the whole system. Interactions mean that the system functions can interact in the software domain as well as in the problem domain. In the case of a computing system, complexity means the time taken in completing the process of execution and the amount of storage required for the computation. If a programmer interacts with the system, then complexity means the difficulty of the programmer in building codes, removing bugs from the code, testing, or maintenance of the program. Complexity increases with interactions. With more modules in a code, the interactions will also be high and hence complexity increases exponentially. Also, if complexity is high then there is possibility of unintended interactions between modules. This can easily cause defects in programming. In the case of an individual program, complexity increases with iterations and loops.

A software system tends to be more complex with the requirements of more functionality. Complexity, once mastered

takes the appearance of simplicity. Complexity in software development can be based on task analysis and synthesis. Task analysis is to break a single complex code into distinct ones which can be later brought together. This is done especially in the case of existing programs. Synthesis involves constructing an artefact to satisfy the requirements. For a simple program, the requirements are always simple and hence synthesis can proceed directly. But for realistic computer based systems, the requirements are always complex and synthesis can proceed only if relevant complexities have been analysed.

For calculating the complexity in software and programs, some measures exist. Some of the metrics are

1. McCabe's cyclomatic complexity metric- Cyclomatic complexity number,
2. Halstead's software metrics,
3. Henry- Kafura's Software Structure,
4. Chidamber- Kemerer's complexity metric for object oriented programs,
5. Branching complexity - Sneed Metric,
6. Data access complexity - Card Metric,
7. Data complexity - Chapin Metric,
8. Data flow complexity - Elshof Metric,
9. Decisional complexity - McClure Metric.

A complexity measure to be successful, the metric development must be based on the theory of programming behaviour. Testing must be performed by keeping a note on the area of applications of the program, and where it can be actually simulated. The complexity analysis methods have been implemented without taking such considerations. Since there isn't a theory of programming behaviour, researchers take conclusions through experimentation. These often seem to be only a weak support for complexity measures.

Literature Survey

Complexity of a program will increase the number of errors in that particular program. If the complexity of a system is higher than expected, then it is more difficult to design, build, and use. The success completely depends on human understanding in the software development. Computational complexity is considered as the total count of steps taken in computing and logical complexity as the total count of steps to verify the perfection of programs. Complexity is a function of the number of test cases handled in the verification process. The logical and computational complexities can be different for the same program. For example, consider the sorting techniques, heap sort and bubble sort.

Bubble sort is having a comparatively less logical complexity but the computational complexity is exceptionally high.Another complexity is the residual complexity. Suppose a program has more logical complexity in the beginning phase. But if it can be used as it is, then the residual complexity is zero. Residual complexity computes the reliability of the software which is modified. If we are adding a new module, residual logical complexity and logical complexity are the same. Residual complexity will be less if the change was expected by the actual design. And if the changes are not expected within the existing architecture then it is difficult to verify. So, it's better to write a new one.The main reason behind complexity of programs is the need for more functionalities and performance. For a better level of performance and features, the developing and using of complex software components are unavoidable in most applications.

Complexity is caused by useful features but not those essential features. We can make use of simplicity to control complexity by distinguishing in between critical properties from desirable properties. Let's consider the scenario of searching an element in an array. The complex property is to find the item correctly. The achievable property is to find it very fast. There are different searching techniques but the fastest is binary search.

Fault-tolerance also adds to the complexity since it demands operation in varying sub contexts within the overall context of the system. Here the problems in the sub context may already have covered in the overall context. Further, complexity increases with different modes of operation. System functions can also interact with other functionalities or problem domain. Feature interaction is another source of complexity and the reason behind difficulty in computer systems. The features of individual behaviours are relatively simple when working separately but when they interact, interference occurs with each other. Such combinations are complex, neither satisfying the individual tasks nor completing the overall task. Software measures are used mainly for deriving a basis for estimates, for tracking progress of the project, to find complexity, for analyzing the defects, for validating best practices experimentally and finally help us to make better decisions. For measuring the complexity of software, mainly cyclomatic complexity is used. Other approach is Halstead's complexity metric.

By using the cyclomatic number definition, McCabe has derived a quantification methodology for software complexity based on the graph theory. In the flow graph he interpreted cyclomatic number as the minimum count of paths. While Halstead quantified using the programs source code.Cyclomatic complexity (Thomas .J. McCabe, 1976) [6] is a software measure used for calculating logical complexity involved in a program. It is a measure of linearly independent paths through the source code of a program. This metric is based on graph theory. By using control flow

graph, developed from the code, in which nodes represent block of code and the edges connects two nodes, if the second block is executed right after the first code. Cyclomatic complexity can also be applied to a separate blocks of code, methods or classes, modules within a program.The basis path testing finds each path which is linearly independent in the program. The Cyclomatic complexity number [6], CCN, of program equals to the number of test cases. For structured programming, if the source code contains no control or decision statements, then the complexity is unity, as there will be only one path through the code. If there is only one IF condition, there will be two paths through the code: one through TRUE part and other through the ELSE part (FALSE condition), so the complexity will be 2. In case of two nested IFs, or single IF with two conditions, then it will be 4.

Hence generated quantitative complexity number will not be dependent on the size of code but influenced by the decision structure of the program or the total count of basic paths through the code. Although complexity can have values from one to infinity, an upper limit has been given by McCabe, i.e. 10. This is in accordance with the value of maximum issues an individual can handle simultaneously proposed by some psychological reviews. It is preferable to give sub-modules or reprogram, if the complexity exceeds 10.

All computer programs can be expressed as graphs i.e. control flow graphs and the CCN can be given mathematically, as

$C(g) = E - N + 2P,$

where

E = the number of edges of the graph.

N = the number of nodes of the graph.

P = the number of connected components.

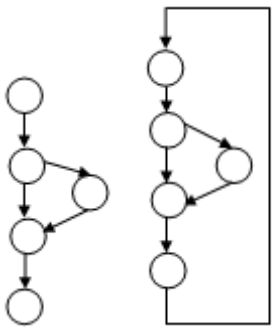

Fig. 1 **Fig. 2**

For example, consider the simple program's control flow graph as in graph Fig.1. The program begins executing and, then enters a loop and finally exits. The above graph has 5 edges, 5 nodes, and 1 component connected, so the

cyclomatic complexity of that program can be given as 5 - 5 + 2*1 = 2. That is there are two paths possible through the graph.

Another method is, for graphs which have every exit point connected back to the given entry point. Then the graph is said to be strongly connected and the cyclomatic number of the code can be formulated as

$$C(g) = E - N + P.$$

Consider the graph in Fig.2, it has 6 edges, 5 nodes,1 component connected, that also results in a cyclomatic complexity of 2 using the other formulation (6 - 5 + 1 = 2). This is calculation of the number of linearly independent loops which exist in the graph. Since every exit point enters back to the point of entry, there will be a minimum of one such cycle for each exit point.

In the case of an individual method or program, P is always unity. The formula for an individual method or function can be given as is

$$M = E - N + 2$$

Halstead's complexity metric

Halstead's complexity metric [9] is an entropy measure which gives the following results:

1. Length can be given as $N = N_1 + N_2$
2. Vocabulary can be given as $n = n_1 + n_2$
3. Estimated length, in case of perfectly structured programs,

   $N' = n_1 \log_2 n_1 + n_2 \log_2 n_2$
4. Purity ratio is $PR = N'/N$
5. Volume is $V = N \log_2 n$
6. Program effort, $E=V/L$,

Where n_1 is the count of distinct operators, n_2 is the count of operands which are distinct,N_1 is number of operators, N_2 is number of operands, $L = V^*/V$ and V^* is the given volume of the design implementation which is compact most.

Chidamber & Kemerer metrics

Chidamber & Kemerer metrics is designed especially for object oriented system development. It is class based metrics which directly calculates the Weighted Methods for Class given as

WMC = $\sum_{i=1}^{n} c_i$, where c_i is complexity of each method in class.

Complexity can be either McCabe complexity or any other metrics. It is better to choose smaller values. But the average complexity per method is the best metric. The following are the parameters taken:

1. Depth of the inheritance tree (DIT) means the length of the highest achievable path from the specific node to the root of the tree. Here, too it's better to choose smaller values.
2. Number of children (NOC) means the number of immediate sub class of the parent class i.e. the count of direct subclasses.
3. With increase in NOC, reuse increases but the abstraction will be diluted. It is always better to have depth than breadth in class hierarchy, because it allows reuse of methods through inheritance.
4. Coupling between classes (CBO) means the count of associations between classes. With increase in CBO, reuse decreases.
5. Method Inheritance Factor MIF = $\frac{\sum_{i=1}^{n} M_i(C_i)}{\sum_{i=1}^{n} M_a(C_i)}$

Where M_i (C_i) means the count of functions that are inherited and which is not overridden in C_i, $M_a(C_i)$ means the count of functions which can be called using C_i and $M_d(C_i)$ means the count of functions that are declared in C_i

$M_a(C_i) = M_d(C_i) + M_i(C_i)$

All the functions which can be requested are the sum of new or overloaded functions and the inherited ones. If MIF is close to 1 then there is little specialization and if MIF is close to 0 then large changes.

Others include Coupling factor, polymorphism factor, etc.

Comparative Analysis

McCabe's complexity metric is used as maintenance metric. Various designs relative to complexity can be given and used as a quality metric. McCabe complexity metrics is faster than Halstead. In McCabe, the minimum effort is measured and it is easy to apply. It also limits the logic of the program during development and guides the testing process.

Whereas Halstead is used for scheduling and reporting projects. So, the overall quality of the program is measured. It depends on completed code. The rate of error and maintenance effort is predicted. Also it is simple to calculate and can be used for any programming language.

McCabe's complexity metrics makes maintenance easier, it's easy to apply, and it limits the complex logic in and hence, can be used during development phase. Halstead metrics gives the rate of error and time and an in-depth analysis of the program is not required. Chidamber & Kemerer metrics is used for object oriented analysis and there

are no thresholds defined for the CK metrics.

However, they can be used identifying outlaying values. Cyclomatic complexity is also related to the lines of code. The graphical representation for different lines of code with cyclomatic complexity is shown below. As the number of lines increases, the complexity also increases.

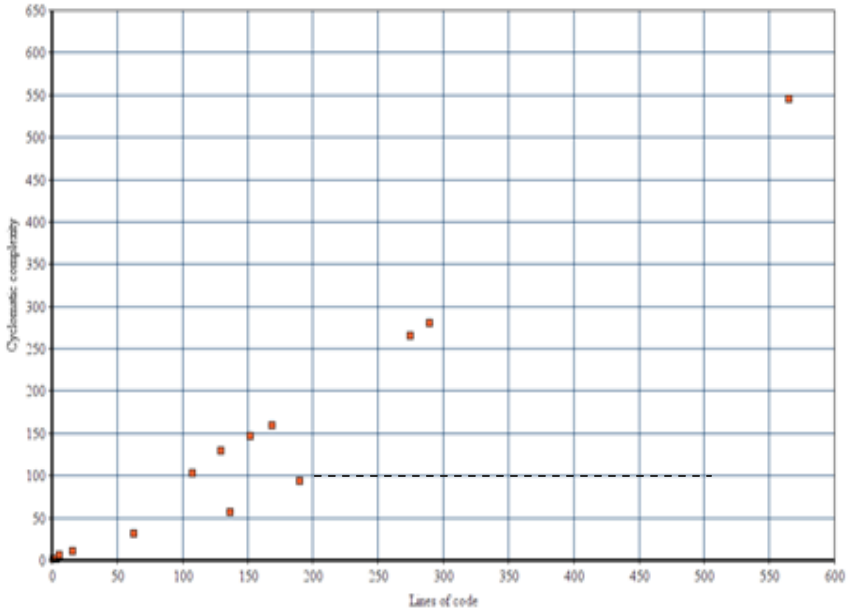

Fig. 3

Simple Integer Program

Consider the graph in Fig. 4 which represents the factorial program using repeated addition as stated by Alan Turing. The complexity metrics can be found as below:

McCabe's complexity: Using the graph in Fig.4, the cyclomatic complexity of the factorial program using repeated addition as stated by Alan Turing can be found. C(g) will be 3, i.e. E – N + 2P = 11-10+2*1. If it's a strongly connected graph with the dotted edge, then also we get the same complexity of 3 i.e. E – N + P = 12-10+1.

Halstead's complexity: The Halstead complexity of the factorial program using repeated addition as stated by Alan Turing would be stated as follows:

Program Vocabulary is $n = n_1 + n_2 = 4+ 6 = 10$

Program length is $N = N_1 + N2 = 12 + 21 = 33$

Calculated length of program is $N' = n_1 \log_2 n_1 + n_2 \log_2 n_2$

$= 4\log_2 4 + 6\log_2 6$

$=23.6174$

Volume, $V = N \log_2 n = 33 \log_2 10 = 109.6227$

Difficulty, D = $\frac{n1}{2} * \frac{N2}{n2} = \frac{4}{2} * \frac{21}{6} = 7$

Effort, E =D * V = 7 * 109.6227 = 767.3589

Time taken to program, T = E/18 = 42.63 sec

Number of delivered bugs, B = $E^{2/3}$ / 3000 = 0.0279

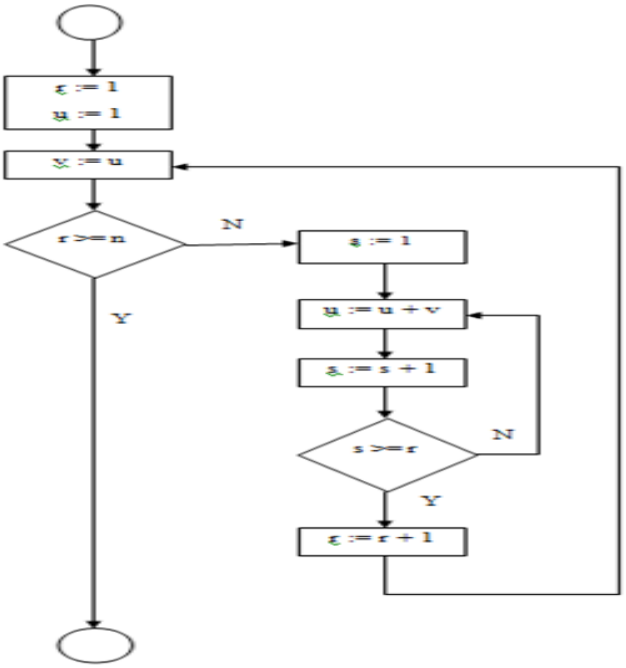

Fig.4

Program with Multiple Traversals

For a comparative study of the complexity metrics, let's consider the program for computing factorial of a program using repeated multiplication as shown in Fig. 5.

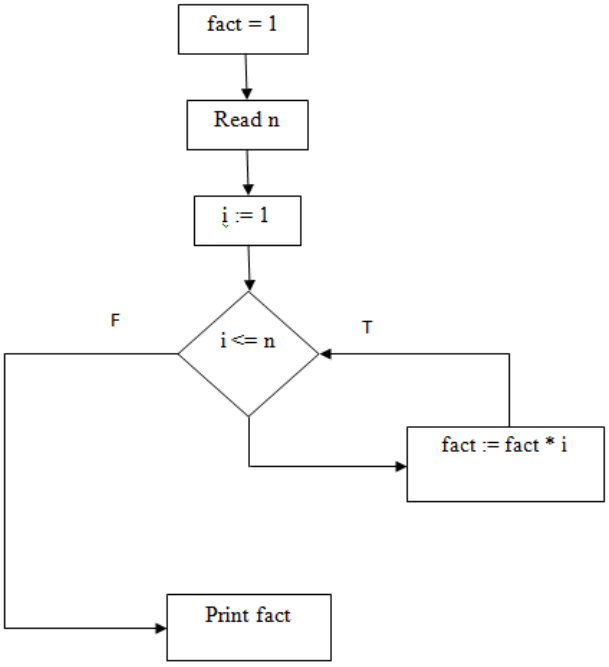

Fig. 5

McCabe's complexity: For the above, the cyclomatic complexity of the factorial program using repeated multiplication can be found as. C(g) = 2, i.e. E – N + 2P = 6 – 6 + 2*1. If it's a strongly connected graph, that is if the last node is connected back to the first node, then also we get the same complexity of 2 i.e. E – N + P = 7 – 6 + 1

Now, comparing this with the complexity of factorial program using repeated addition, we can see that here the complexity is reduced. This clearly states that multiple traversals can reduce the program complexity. For further proof, we can go for Halstead's complexity.

Halstead's complexity: The Halstead complexity of the factorial program using repeated multiplication can be given as:

Program Vocabulary is $n = n_1 + n_2 = 5 + 4 = 9$

Program length is $N = N_1 + N_2 = 7 + 11 = 18$

Calculated length of program, $N' = n_1 \log_2 n_1 + n_2 \log_2 n_2$

$= 5\log_2 5 + 4\log_2 4$

$= 19.605$

Volume, $V = N \log_2 n = 18 \log_2 9 = 57.05865$

Difficulty, $D = \frac{n1}{2} * \frac{N2}{n2} = \frac{5}{2} * \frac{11}{4} = 6.875$

Effort, $E = D * V = 6.875 * 57.05865 = 392.2782$

Time taken for programming is $T = E/18 = 21.79323$ sec

Number of bugs delivered is, $B = E^{2/3} / 3000 = 0.01786$

Comparing the length of the program, vocabulary, volume, difficulty, effort, the time taken for program and the number of bugs delivered of the two programs, the values in multiple traversal programs is reduced. This means that instead of a complex traversal, it is better to go for multiple simple traversals which can reduce the program length, time consumption, difficulty level and even the bugs.

Conclusion

As the software industry advances, the complexity of codes is of utmost importance. Hence, the need for better software metric keeps on increasing. The complexity metrics has a major role in developing simple and maintainable software than complex ones. Using a single complexity metric, we can't determine the complexity of the program or software system in all aspects. So, it's better to use more than one type of complexity analysis for studying the complexity of programs. Also, based on what criteria in which we have to improve our program, we

can choose a complexity accordingly. Considering all these factors, the software development process can be easier and the resultant software will be robust, simple and maintainable. As a result software can be reused and the underlying cost involved in maintenance can be reduced.

References

1. M. Jackson, "Simplicity and complexity in programs and systems," Conquering Complexity, Springer-Verlag London Limited 2012, Chapter 2, pp. 49-72.
2. Thomas J. Walsh., "A software reliability study using a complexity measure," National Computer Conference, 1979, pp. 761-768.
3. Lui Sha, "Using simplicity to control complexity," Fault tolerance, IEEE software, July/August 2001, pp. 20-28.
4. Chapid, Ned, "A measure of software complexity," National Computer Conference, 1979, pp. 995-1002.
5. John L. McTap, "The complexity of an individual program," National Computer Conference, 1980, pp. 767-771.
6. McCabe, Thomas J., "A complexity measure," Software Engineering, Volume SE-2, Number 4 (December 1976), pp. 308-320.
7. McClure, Carma L., Formalization and Application of Structured Programming and Program Complexity, Ph. D. Thesis Illinois Institute of Technology, Chicago, 1976,288 pp.
8. McClure, Carma L., Reducing COBOL Complexity through Structured Programming (New York: Van Nostrand Reinhold, 1978), 192 pp.
9. Halstead, Maurice H., Elements of Software Science, New York, Elsevier North-Holland, Inc., 1977, 127 pp.
10. Hanson, Wilfred J., "Measurement of program complexity by the pair cyclomatic number, operator count," SIGPLAN Notices, Vol. 13, No. 3, March 1978, pp. 29-33.