THEORETICAL APPROACHES IN SOFTWARE COMPLEXITY METRICS

5. *C. Chiemeke & A. O. Oladipupo*

Abstract

This paper presented a review of some of the theoretical issues involved in software complexity metrics. The paper was divided into four parts. The first part attempted to consider the meaning, forms and levels of software complexity. The second part examined the definitions of metrics and presents a review of the existing software complexity metrics. The properties of software complexity metrics were considered in part three while the last part is on the conclusions.

A review of the existing work on software complexity metrics shows the use of McCabe's cyclomatic number, Halstead's software science, the number of program statements and the Oviedo's dataflow complexity graph. None of these metrics satisfied all the desirable nine properties of a complexity metric. The conclusion was that the choice of the complexity metric is dependent on the choice of the software development methodology and the programming language. The quest therefore was to see how the choice of software development methodology and programming language can help to minimize the software complexity.

Introduction

The main task in software engineering is to design and develop workable computer-based solutions for problem-solving tasks. The workability and reliability of the emerging computer-based solutions depends on the evolving software. The workability and reliability of the software depends inter-alia on the understanding of the original problem by the software engineer. Similarly, the software engineer's understanding of the original problem also depends on how the problem is structured among other things.

All things being equal, a simple and well articulated and structured problem (task) will promote software engineer's full understanding of the problem and lead to articulation of appropriate and complete solution, and vice versa. Faults in Software development may arise from design phase i.e. design errors, improper modification or transient hardware errors which corrupt the stored program. It is often argued that errors in Software design most often arise from the complexity of the problem. Woodfield (1989) observed that 25% increase in complexity of the problem would lead to 100% increase in emerging program or software complexity. High risk and excessive cost are associated with poor-equality and complex software design (Markusz and Kaposi, 1985). Maintenance of large-scale complex software systems is a very expensive, time consuming and error-prone activity. Research interest has emerged on measurement of software complexity as a means of understanding how to control the degree of complexity of software design and development.

Software complexity metrics seek to measure the degree of error-proneness, the ease of debugging and modification of the software development process (Potier et, al. 1982. Rault 1979; Schnerdewind and Hoffman, 1979). There are many factors that software designers would put into consideration while seeking for a computer-based solution to a problem. These include performances, reliability, and availability of the requisite resources, adaptability, maintainability and complexity. Of all these factors complexity impinges on many of these other factors especially maintainability and dependability or reliability. The aim of software complexity metrics therefore is to seek for ways of reducing faults in software design and development and to promote the efficiency, reliability and maintainability of software. This implies that there is much more to good software design than mere knowledge of programming language. Therefore structured programming and other various software design methodologies seek to control software quality by imposing a discipline on the designer who then controls the complexity of design tasks and supplement the rules of the programming languages (Markusz and Kaposi, op cit).

Curtis et al (1979) identified two forms of software complexity as computational and psychological complexities. Computational complexity is the complexity of the program for problem-solving task. It is the quantitative, aspects of the algorithm meant for solving a given problem, ft relates to the speed of the execution of a program. However, the measurement of the difficulty of the processes of design, comprehensive maintenance and program modification form

the corc of psychological complexity. Psychological complexity relates more to the complexity of the problem-solving task to which the designer is seeking a computer-based solution. The difficult in measuring (psychological) complexity of problem is due to the fact that the concept of a "problem" is too vague. Hence, emphasis is usually on the program complexity. program complexity can be assessed at two levels, namely the internal and external c o m p l e x i t y . I n t e r n a l complexity of software is determined by the kind of interaction that exists within each module of the program. While the amount of interaction a module has with its environment, which is defined by other modules, is the measure of the external complexity of the program. Lew et al (1988) conclude that the overall complexity of each module and the external complexity due to modul interrelationships.

Ramamoothy et al (1985) also identified two levels of software complexity as sequential complexity and complexity due to concurrency. They argued that program complexity of a distributed program, which realises concurrency by parallel execution execution of separate task communication, would consist of local complexity and communication. Local complexity is the complexity of the individual tasks without taking cognizance of their interaction with other tasks. However, the complexity arising from the interaction amongst the different task is the communication complexity. Hence, the classification of program complexity by Lew et la is akin to the classificatory scheme provided by Ramamoorthy et al (1985).

Kafura and Reddy (1987) related cyclomatic complexity and six other software complexity metrics to the experience of various maintenance activities performed on a database system consisting of 16,000 lines of FORTRAN code. The authors used a subjective evaluation. technique, relating the quantitative measures obtained from the metrics to assessments by informed experts who are very familiar with the system being studied. This was done to determine information obtained from the metrics would be consistent with expert experience a as a guide to avoid poor maintenance work. They analysed changes in system level complexity for each of three versions. The authors observed that the "change in complexities over time agrees with what one would expect". It was also observed that the complexity growth curves "seemed to support the view that maintenance activities, either enhancements or repairs, impact many different aspects of the system simultaneously". It was also observed that changes in procedure arnrnpil could be monitored to question large changes in complexity that is not planned They that is not planned. They noted that developers tend to avoid changing very complex procedures during maintenance, even if the alternative changes lead to degradation in overall design quality, and suggeste: that uumi monitor the implementation of those changes.

Definitions of Metrics

The earliest attempts at measuring software complexity dated back to the work of ***Dijkstra*** (968). Dijkstra (op. cit) observed that the quality of program is a decreasing function of the density of *go to statements* found in it. Hence, a very simple measure of software complexity is the number t*os* in a program. This was observed as an inappropriate measure of unstructr.rdncss far languages like FORTARAN, but can be used for ALGOL type of language. However,it had noted the *go los* are essential ingredients for writing structured FORTRAN (Neely, 1976)

Gilb (1977) observed that logical complexity is a measure of the degree making decision within a system and that the number of *If statements* is a rough measure of this complexity Farr Zagorski (1965) found this metric to be a significant factor in predicting software cos:, h wm'opi tht it is reasonable generally to count not only the ifs but also the entire branch creating statement (decision points) in a program.

McCabe (1976) adopted a different approach to software complexity measure developed a theory based on the modelling of programs as directed graphs. The complexity of the program was then said to be measurable by the cyclomatic complexity of the digram The original formulation of McCabe's theories was said to lack rigour.

Kaposi et al (1979) measured the complexity of PROLOG program. PROLOG is a language based on first order predicate logic, in which the specification of the problem and the means of realising the solution can be expressed. The rules of the PROLOG language demand the explicit statement of the problem on hand, and the composition of the solution as a s***trict hierarchical*** structure of related and explicitly specified parts called partitions. Each partition could be considered.

as an autonomous entity; hence the complexity of the designer's task could be related to the local complexity of partitions rather than to the global complexity of the PROLOG programs as whole.

The complexity of a partition appears to depend on the data relating it to its environment, the number of subtasks within it, the relationships among subtasks, and the data flow through the structure. Local complexity was then expressed as a function of these four arguments as parameters. The complexity functions was considered as the unweighed sum of complexity parameters as follows:

$$X = Pi + P2 + P3 + P4 \quad \text{.....} \quad (1)$$

where. PI = No of new data entities in the positive atom of the partition, in the problem to be solved by the partition.

P2 = No of negative atoms in the partition, i. e. the number of sub- problems with which the problem divides;

P3 = The measure of the complexity of the relations between negative atoms:

P4 = No. of local variables linking the sub-problem.

The global complexity of a PROLOG program is the unwieghed sum of the local complexity of all n constituent partitions i. e.

$$g = \sum_{i=1}^{n} X_i$$

where, X_i = local complexity of i^{th} partition.

The partitions were sorted into four complexity bands according to the value of their complexity functions as follows;

Range	Complexity Bound	Complexity code of partition
$1 < X < 3$	Trivial	T
$i < X < 7$	Simple	S
$8 < X < 17$	Complex	c
$X > 18$	Very Complex	V

Table I. Complexity Bounds

Later development showed a deficiency of the complexity function of the equation formulated by Kaposi et al (op. cit) in capturing all of the aspects of task complexity. Thus, Markusz and Kaposi (1985) proposed the introduction of new parameters into the complexity function of the earlier equation. The proposed complexity function then become.

$$X = P\backslash + P2 + P3 + P4 + f1 + f2 \quad \text{.....} \quad (2)$$

Where.

PI - P4 = number of new sub-problems and the relations between them

f 1 - f2 = number of new data entities and relations between them

f2 = number of different operators (infix, or prefix) within the partition

f2 = measure of the complexity of the relationships among the new data entities.

From their experiment they found out the new measure represents a consider P improvement in quantifying the difficulty of design tasks. They concluded by saying that : t complexity measures proposed could be applied in two ways:

(i) to prevent errors. Controlling the quality of designed software as a means of qua assurance, to detect the areas of potential design weakness in existing software: and

(ii) to guide the process of reconstruction into functionally equivalent but compievr. controlled firm.

Shatz (1988) proposed complexity measure for distributed software using Ada language programming language for implementation. Reviewing the previous work of complexity metric Ada (see Gannon et al, 1986; Bombach and Basili, 1987): the author observed that none of the existing research then has explicitly considered the concurrency features of Ada languages. Shatz (op. cit) opined that for distributed program, which realises concurrency by parallel execution separate tasks and which constrain the concurrency by introducing task communication, the pros'a- complexity consists of two components, local complexity (LC) which reflects the complexity of re individual tasks, disregarding their interactions with other tasks; and communication complex r. (CC) which reflects the complexity of the interactions among the tasks. Hence, a distribute: program's complexity (TC) was then formulated as:

$$TC_{,} = W * I(LQ) + X * CC \quad \sum_{i=1}^{T}$$

Where:

T	Number of tasks
LCi	the "local complexity" of task i.
W and X	are weight values used to adjust for the fact that the two types of complexity many not carry equal weight in deciding the overall complexity.

The communication complexity metric was based on counting the number cf communication statements in the program. For Ada program, this implies counting the number cf Entry call statements and Accept statements. Hence, this metric was found to correspond to McCabe's cvclomatic metric, which counts decision points. For local complexity the knot, count metric, which examines the relations between the decision points, was introduced.

Lew et al, (1988) realised the existence of many internal complexity measures, which have proved useful in software design. The author than aims at developing an external complexity measure. An external complexity metric of a program measures the amount of interaction between a module and its environment. This is expressed as

$$Ci = I\,(TICy + TICj)$$

Which is the sum of interaction between module i and the other modules j, i + j; Where:Ci = external complexity of a module i

TICij = total information content of the message from i to j. and

TICji = total information content of the message from i to j.

However, the overall complexity of a system is made up of both internal complexity of each module, and the external complexity due to module interrelationship. Thus, the complexity of a module (M_f) in a system is the weighted sum of the internal and external complexities Kj and Cj respectively, i.e.

Mi = (a * Kj) + (b * C;)

Where: a and b = weighting factors, and
a + b = I.

Properties of Metrics

In attempt to evaluate software complexity measures, Weyuker (1988) outlined a set of nine desirable syntax properties. Four of the complexity measures were subject to evaluation desirable syntax properties. Four of the complexity measures were subject to evaluation based on the properties. This tested complexity measuring include the number of program statements, McCabe's cyclomatic number, and Halstead's programming effort and knot program is zero, as use was made of structured programming language for implementation. The statement counts as a measure of program complexity is the number of program statements. This was considered a very simple way of computing program complexity. The McCabe's cyclomatic number defines the complexity of a program as

V = e - n + 2p

Where:

e = number of edges in a program flow graph

n = number of nodes

p = number of connected components.

However, if p = 1. Then V =)(1

Where)(= numbering predicates in a program.

Halstead's programming effort in defining the program complexity first of all defined the following terms:

(a) program volume (V) as

V = (N| + N2) log2 (m + n_2)

Where:

ii| = number of distinct operators ***n_2 =***

number of distinct operands ii| =

number total number of operators n_2 =

total number of operands.

(b) Potential Volume (V)

(V) = minimum possible volume for a given algorithm

(c) Programming Effort (E)

E = V^2 / V*

Since V* is difficult to compute, therefore program complexity (E). which measure, is then estimated as follows:

$$E = \frac{= n, N_2 (N, + N_2) Iog_2 (ii| + n_2)}{n_2}$$

Note that: E (P) = Effort for program body P
”2 (Q) = Number of distinct operands in Q

Oviedo method of complexity measure makes use of the dataflow c h a r a c t e r i s t i c s program. Although GO TO statements are permitted in his language, they do not contribute dataflow complexity of the program.

The properties of complexity measures applied 011 the four methods of complexity metrics outlined above include the following ;

Property 1: (3P) (3**Q)|p|≠** |Q|)

This implies that a measure, which rates all programs as equal!;. corT.piex. ins not really a measure. It was found out that all the four complexity mezsincs under consideration satisfied this property. It has been pointed out ti"3t airr

all

measures have an infinite spectrum of values, property one appears t: Se redundant.

Property 2: complexity measure should not be too coarse. A complexity measure u :ranr if it is not sensitive. A measure is not sensitive enough if it divides .1 programs into just “a fair'’ complexity classes.

Only the statement count fulfils this property while cyclorr.av.: mlkioir and dataflow complexity do not fulfil property 2.

Property 3: there are distinct programs P and Q such that |P| = |Q|
The first three properties above are properties of measures, which do not distginslish between syntax and semantics.

Property 4: 0P)(3Q) (P = Qand|PMQ|)
This implies that the program’s complexity depends on the details of the implementation. All the measures satisfy this property because the measure are entirely implementation dependent.

Property' 5: (Ⱥ) (ȺQ) |P | ≤ |P ; Q | and | Q | ≤ | P ; Q |)
Both the statement count and cyclomatic number satisfy these relati while the dataflow complexity and the effort measure do not.

Property 6: (a) : (3 P) (3 Q) (3R) (| P | = | Q |) and | P ; R | ≠ | Q ; R |)
(b) : (3 P) (3 Q) (R) (| P | = | Q |) and | R , P | ≠| R ; P |) '

Property 7: Program complexity should be responsive to the order to the statements, ani the potential interaction among statements. Neither statement count, cyclomatic number and effort measure satisfy the property.

Property 8: If P is renaming Q , then | P | = | Q |
This property is satisfied by each of the considered complexity measure:

Property 9: (3 P) (3Q) (| P | + | Q | < | P ; Q |)

In conclusion tlie authors concluded that none of the four software complexity metrics satisfied all of nine properties but that the dataflow complexity and effort performed best.

In a subsequent paper, Cheriniavsky and Smith (1991) presented a complexity measure, which satisfied all of the nine principles. However, the complexity measure lacks absolute practical utility in measuring the complexity of a program. The paper concluded by saying that, at best, satisfying all of the nine properties is necessary, but not sufficient, condition for a good complexity measure. The authors used the same programming language described by Weyuker to introduce a complexity measure, which satisfies all nine properties.

The research on the software complexity measure is still in progress. One looks towards a single metric of complexity that will both fulfill all the properties of a desirable metric and still have practical utility in measuring software complexity irrespective of the programming language utilised for implementation.

Conclusion

The importance of metrics has been recognised because without metrics the tasks of planning and controlling software development and maintenance will remain stagnant. With metrics, software projects can be quantitatively described, and the methods and tools used on the projects to improve productivity and quality can be evaluated. The issues in software complexity metrics involve selecting a metric and ensuring that the metric measures whatever it is expected to measure. There are a number of metrics on complexity in software design and development. Some are for internal complexity measures while others are for external complexity measures. A review of the existing software complexity metrics shows that none is completely adequate in meeting the requirement of the designers. It has been argued that the software development methodology adopted has impact on the level of software complexity obtained. Take for example; the top-down development methodology should be used for Halstead's measure to achieve low complexities while structured programming technology should be used for McCabe's metric to achieve low complexities.

fhe search for an absolute metric of software complexity may be futile but whatever emerges as a reliable software complexity metric may be software development methodology and programming language dependent. Hence, the search continues.

References

Cherniavsky, J. C. and Smith, C. H. (1991) Concise Papers on Weyukers Axioms for Software Complexity Measures". ***IEEE*** Trans, on Software Engineering Vol. 17, No. 6.

Curtis, B. Sheppared, S. B.; Milldam. F.; Bort, M. A. and love T, (1979). Maintenance Tasks with the lialstead and McCabe metrics. ***IEEE Tran. In software Engineering*** SE-5 (2).

Dijikstra, E. W. (1968), Goto Statement Considered harmful". ***Comnntn. Ass. Comput.*** Mach., Vol. 11 pp. 147 - 148.

Farr, L. and Zagorski H. J. (1965) Quantitative Analysis of Programming Cost Factors: A Progress Report, in Economics of Automatic Data Processing in ***1965 ICC Sypop. Proc***., Rome, A. B. Fried link, Ed. Amsterdam. The Netherlands: North - Holland.

Gilb, T. (1977) Software Metrics. Cambridge, M. A: Winthrop.

Kafurn and Reddy (1987) The Use of Software Complexity Metrics in Software Maintenance. ***IEEE Transactions oft Software Engineering*** SE - 13 (3), 335-343.

Kaposi, A. A.; Kassovitz, L. and Markusz. Z. (1979) PRIOLOG a Case for Augmented PROLOG Programming Proc. Informatica, Bled, Yugoslavia.

Lew, K. S.: Dillon, S.T. and K. E. (1988). Software Complexity and its Impact on Software Forward Reliability ***IEEE Trans, on Software Engineering*** Vol. 14, No. II.

McCabe, T. J. (1976) A Complexity Measure. ***IEEE*** Trans. Software Engineering Vol SE- 2, pp. 308 -320.

Markusz, Z. and Kaposi, A.A. (1985). Complexity Control in Logic-based Programming ***The Computer Journal*** Vol. 28. No. 5.

Neely, P. M. (1976) The New Programming Discipline. ***Software - Practice and Experience,*** Vol. 6. Pp. 7-27.

Potier, D; Albin, J. L.; Ferreol, R. and Bilodeau, A. (1982) Experiments With Computer Software Complexity and Reliability in ***Proc. 6th Nil. Conf Software Eng.,*** PP 94 - 103.

Ramamoorthy, C. V.: Tsai, W. T.: Yamaura, T. and Bhide, A. (1985) Metrics Guided Methodology, in ***Proc. 9th Computer Software and Application Conf.,*** P. 111.

Rault, J. C. (1979) An Approcah Towards Reliable Software in ***Proce. 4'th Nit. Conf. Software Eng.,*** pp. 220-230.

Shatz, S. M. (1988) Towards Complexity metrics for Ada Tasking IEEE Trans, on Soft. Eng., Vol. 14, No. 8.

Schneidewind N. F. and Hoffmann, H. M. (1979) An Experiment in Software Error Data Collection and Analysis ***IEEE Trans, on Software Eng.*** Vol. SE-5, No 3, pp. 276-286.

Woodward, M. R.; Hennel, M. A. and Hedley, D, (1979) A Measure of Control Flow"' Complexity in Program Test IEEE Trans. Software Eng. Vol. SE- 5, No. 1, pp. 45 — 50 .

Woodfield. S. N. (1979). An Experiment on Unit Increase in Program Complexity ***IEEE Trans. Software Eng,;*** Vol. SE - 5, No. 2, pp, 76 - 79.

Weyuker. E. J. (1988) Evaluating Software Complexity Measures. ***IEEE Trans, on Software Eng.,*** Vol. 14, No. 9.