ISSN (Online) 2278-1021
ISSN (Print) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 4, Issue 4, April 2015

Analysis and Evaluation of Quality Metrics in Software Engineering

Zuhab Gafoor Dand[1], Prof. Hemlata Vasishtha[2]

School of Science & Technology, Department of Computer Science, Shri Venkateshwara University, U.P, India[1]

Associate Professor, Department of Computer Science, Shri Venkateshwara University, U.P, India[2]

Abstract: In this research paper the study of various software metrics are performed which are classified into three categories: primitive, abstract and structured. The study is performed to provide software developers, users and management with a correct and consistent evaluation of a representative sample of the software metrics available. The analysis and evaluation of metrics was performed in an attempt to: assist the software developers, users and management in selecting suitable quality metric(s) for their specific software quality requirements and to examine various definitions used to calculate these metrics.

Keywords: Software Quality Matrices and Software Engineering

I. INTRODUCTION

The research in the area of software quality metrics generally and software quality particularly is still in a state of flux. This is because the area is in its infancy. There is a great demand for ideal software quality metrics and a real need for distinct and precise definitions of software quality and related terms. Software metrics can be considered as the means of measuring software qualities. These measurements are required for quantitative comparison, cost estimation and quality evaluations. The term software quality, for which various metrics have been developed and applied, is one of those terms in Software Engineering which has not yet been defined clearly, precisely and properly despite numerous attempts. Moreover, other terms related to software quality such as criterion, factor, characteristic, attribute, etc., are also not defined precisely and clearly by Software Engineers. For example, Hocker et al [1] use the terms attribute and criterion for the same thing. The term quality characteristic is used by McCall et al [2] as a quality factor. Kitchen ham et al [3] uses the terms quality factor, quality attribute and quality characteristic with the same meanings. Further, the term quality is used by Jones [4] to denote the absence of defects from the software without defining the term quality. Yourdon [5] has used the term quality without having any definition. He considered a high quality software system design, as one which consists of modules having a high degree of functional cohesion.

The IEEE standard glossary has defined the term software quality. In their definitions, they have used certain terms or words without giving any definition. For example, the terms "characteristics" and "attributes" have been used without giving a definition in the glossary. This confuses the users, management and developers. Boehm et al [6] have not given any specific definition of terms software quality and software quality characteristic when they were discussing and developing their hierarchy of software quality characteristics. A study similar to Boehm's approach was carried out by McCall et al [2] to define software quality aspects. They have defined quality as: "a general term applicable to any trait or characteristic,

whether individual or generic; a distinguishing attribute which indicates a degree of excellence or identifies the basic nature of something". In the above definition, the authors have not given any definition for the terms characteristic and attribute which are used in their definition. Later during the study of software metrics, it was observed that, up to now, efforts in developing software quality metrics have been concentrated on very few quality attributes such as complexity, stability, etc. On the other hand, for certain important quality attributes such as usability, readability, etc., real metrics are still not available. Moreover, some desired attributes of software quality can only be satisfied at the expense of other attributes. The evaluation of the available software metrics is mainly needed to select a suitable candidate metric (s) which can be used as a measurement, estimation and forecast of the quality of a software system.

To evaluate the available software metrics, a set of criteria of goodness is essential. Halstead [8] has developed software metrics which have received much attention in literature of software science. These metrics are based on counting the lexical tokens in the program. Many attempts have been made to create quantifiable control flow complexity metrics such as McCabe [7], Woodward [9] etc. These metrics are based on graph theory. Further many other researchers such as Henry et al [10], Haney [11] etc., have developed a number of software structured metrics.

These metrics are based on system component connections. There are a large number of software metrics available, but the difficult problem is, how to evaluate and select a suitable and reliable one. In this study, software metrics are classified into three categories so that an analytical comparative evaluation of the available metrics can be carried out easily. These categories are:

1) Primitive software science metrics which are based on counting lexical tokens of a program.
2) 2. Abstract software metrics which are based on graph theory.
3) 3. Structured software metrics which are based on software system component connections.

II. LITERATURE SURVEY

Metrics for software products can be classified into the following categories:

PRIMITIVE SOFTWARE METRICS

The primitive metrics are based on counts of lexical tokens in a program or program interface features. This type of metric can be applied during the implementation phase of the software development life cycle.

Halstead's Metric

Halstead [8] is the first who presented lexical analysis in his theory of software science. He argued that algorithms have measurable characteristics analogous to the physical laws. In a given program he counted the number of unique or distinct operators (=n1), unique or distinct operands (=n2), total usage of all of operators (=Nl), and total usage of all of operands (=N2).

Albrecht A.J Function Points Metric

Albrecht A.J. [12] has developed a metric called the function points metric. His metric is in the same class as Halstead's metric [8], but instead of counting operands and operators as in the Halstead case, Albrecht counts the number of external functions in the program. Albrecht's function points metric is developed to estimate the complexity of a function which a program performs in terms of input and output data. The general approach is to list, and count the number of external user inputs, inquiries, outputs, master files, and interfaces to be delivered by the development project. Albrecht A.J [12] has pointed out that "these factors are the outward manifestations of any application. They cover all the functions in an application". These counts are weighted by numbers so as to reflect the function value to user/customer. The weights used were determined by Albrecht through debate and trial.

ABSTRACT METRICS

The abstract metrics are based on graph theory. In this category for example the researchers measure the properties of a program control flow diagram. The researchers have developed metrics which are derived from a flow graph representation of a program and use these to show the difficulty of carrying out tasks such as coding, debugging, testing and modifying software. This type of metric can be applied to the implementation phase and to the design phase of the software development life cycle.

McCabe's Metric

Thomas McCabe [7] has developed a complexity metric which is based on the control flow graph representation of a program. The control flow graph is defined as a directed graph in which each basic block of the program is represented by a node, and the possible flow of control between these blocks is represented by an edge. McCabe's metric is denoted by V(G), and is computed as:

$V(G) = E-V+2P$ where;
E is the number of edges in the flow graph representation of the program,

V is the number of nodes in the flow graph representation of the program,

P is the number of connected components in the flow graph.

McCabe has recommended that: the upper bound of complexity, V(G) in any particular module should have a maximum value of "10". If the complexity of a module is greater than "10", then the module should be decomposed or recoded. This enables a software engineer to control the size of a program by setting an upper limit to the measure instead of using just physical size. If a program is decomposed into m connected components, then the value of McCabe's metric for that program will be the sum of the cyclomatic complexities of the components calculated as:

$V(G) = \sum_{i=1}^{m} v(Gi)$ where,

V (Gi) is the complexity of the individual modules

McCabe showed that the cyclomatic complexity of any structured program is equal to the number of predicates in the program plus one. Applying McCabe's metric to the reduced graph G' gives a measure which is termed the essential cyclomatic complexity. McCabe's essential cyclomatic complexity number, EV (G')-l for a structured program.

Knot's Metric

The Knot's metric is a measure which has been proposed by Hedley Knots's metric is based on control flow the actual sequential source program of structuredness Woodward et al [9]. Their edges drawn on The Knots measure are denoted by K and it is calculated as the number of unavoidable crossing points of the control flow edges. This is defined mathematically by Woodward et al [9] as: if a jump from ordered pairs of line A to line B is represented by the integers (A,B) then the jump (X,Y)causes a Knot to occur, if any of the following two conditions is satisfied:

1) $MIN\{A,S\} < MIN\{X,Y\}, < MAX\{A,S\}$ AND $MAX\{X,Y\} > MAX\{A,S\}$
OR
2) $MIN\{A,S\} < MAX\{X,Y\}, < MAX\{A,S\}$,AND $MIN\{X,Y\} < MIN\{A,S\}$.

Woodward et al [9] used the idea of control graph reduction to define what they called the "essential knots" of a program. A control flow graph of a computer program can be represented as a directed graph. Such a directed graph can be reduced by replacing the sub graphs corresponding to admitted primitives of structured programming by a single node. Woodward et al [9] stated that" a structured program will be reducible to a single node with zero knots. This leads to a definition of the remaining knots as the essential knots of the program".

STRUCTURED METRICS

The structured metrics deal with measuring the structured properties of software design. Many authors have described how to measure or assess the qualities of a system by measuring properties such as connectivity and cohesion. Yourdon [13] has used cohesion and coupling as metrics to measure the qualities of modules. Such metrics can have significant impact on the software design and development task. This is because structured metrics can be taken early in the life cycle of software system development task [14].

Haney's Stability Metric

Haney [8] has developed a metric for determining the stability of large systems which depends on module connections. Haney's metric is based on the assumption that the intermodule connections are the main causes of high cost and delayed delivery dates. These types of problems usually occur in systems where modules are heavily connected (highly coupled) and any change to a single module causes subsequent changes in most of its connecting modules. The system's resistance to such changes is called system stability. Haney assumed that a system consists of n modules and Pij is the probability that a change in module i induces a change in module j. Further, with each module i, there is an associated number N which is the number of changes that must be made in module i upon the integration with the system. The probability that a change to module i propagate to module j in two steps is given by:

$\sum_{k=1}^{n}$ Pik Pkj

Which represents the sum of probabilities that a change in module i is propagated to module k and then to module j. In general, the (ij)th element of the probability matrix P raised to the kth power represents the probability that a change in module i will propagate to module j in k.

Myer's Metric

Myer, G.J., [15] has developed a structured metric which depends on the degree of interdependence among the components of a program. The major step in calculating this metric is to develop a complete dependence metric (COM) which describes all dependencies among all modules. Once such a matrix is obtained, the following can be determined easily:

1) The summation of all elements in the matrix divided by the dimension of the matrix (no. of the modules) can give the expected number of modules that must be changed when any single module is changed. The same metric is used by Myers [16] to compute the complexity of the overall program.
2) The summation of all elements in any row i in the matrix can give the expected number of the modules that must be changed when module i is changed.

The first order dependence matrix is derived using the following steps.
1) Evaluate the coupling among all of the modules in the program.
2) Construct an M*M coupling matrix, C, where; M denotes the number of modules in the program.
3) Evaluate the strength (cohesion) of each module in the program.

S. Henry and D. Kafura's Metrics

Structured metrics based on information flows have been developed by S. Henry and D. Kafura [10]. Their metrics measure:

1. **Procedure complexity**: The procedure complexity depends on two factors;

a. the first factor is the internal complexity of the procedure. This is based on counting the number of lines of code in the procedure,
b. the second factor involves the complexity of the procedure's connections to its environment. This involves the information flow connections of a procedure to its environment. The information flow can be determined by the fan-in and the fan-out. The fan-in and the fan-out represent the total possible number of combinations of an input source to an output destination. The whole procedure complexity is computed as:

P(C) - L* (fan-in*fan-out) **2 where
L is the procedure length expressed in lines of code, fan-in is the local flows into a given procedure plus the number of data items from which the procedure reads, fan-out is local flows coming out from a given procedure plus the number of data items to which the procedure writes.

According to Henry et al [6], the local flow of information from module A to module B occurs if one or more of the following conditions hold:
1. If module A calls module B,
2. If module B calls module A and A returns a value to a, which a subsequently utilizes, or
3. If module C calls modules A and B passing an output value from A to a.

The above procedure complexity can be helpful in locating the stress point the procedure with heaviest data traffic) of the software system.

2. Module complexity metric

The module complexity is computed as the sum of the complexities of the procedures within the module. Where a module is defined as: with respect to a data structure D the module consists of those procedures which either directly update 0 or directly retrieve information from D [10].

3. Interface complexity
Interface complexity depends on two factors; the interfaces which connect the system components, and the number of information paths used to transmit information between the components of the system.

III. EVALUATION OF THE SOFTWARE METRICS

There are very many software metrics available, and it is a difficult problem to evaluate and select a suitable and reliable one. Most of these metrics measure the complexity attribute of a control flow graph of a computer program. Baker et al [17] chose three software metrics for the purpose of comparing and evaluating them. These metrics were Halstead's metrics [8], McCabe's metric [7], and the Knot metric [8]. Baker et al have showed some basic properties of each of them.

However, the drawback of their approach was that they did not consider all the necessary aspects of these metrics such as the validity, applicability, etc. Therefore Baker et al descriptions are not enough to select a suitable metric. Further, they did not mention any thing about the structured metrics which are very important at least for the management of software systems.

In the following section a set of detailed criteria of goodness are given against which, each software metric can be evaluated.

General Criteria of Goodness
General criteria of goodness are those criteria which can be applied to any metric for the purpose of evaluation. In this research paper we are using three basic criteria which are explained below.

Applicability: the term applicability means the suitability of metric(s) to the output of different phases of the system development life cycle, i.e., number of the software life cycle phases in which the metric under study can be applied.

Modularity: Modularity is derived from module and it can be defined as: the extent to which a software system can be decomposed into modules provided that a change in one module has a minimal impact on other module.

Modularity can be achieved by isolating frequently occurring sequences of duplicate code [18]. It is worthwhile knowing how much the modularization issue can affect the value of the software metrics.

Language Independency: language independency means the software metric(s) should be computable for any software system, independent of the language in which the system is written.

Comparison between the present metrics
The criteria of goodness were generated after performing a comprehensive study of a selection of the most popular metrics.

PRIMITIVE SOFTWARE METRICS
Halstead metrics have received considerable attention in the software literature, so in this portion we apply the above criteria of goodness, to Halstead metrics.

Applicability
Halstead's metrics which are based on collecting the lexical tokens in a program are applicable to the implementation phase of the life cycle. This is accepted by Halstead [8]. According to Baker, et al [17] the software science metrics are generally developed to measure overall program complexity. It would be possible to apply Halstead's metrics to the output of each phase of the life cycle provided that at each phase notations can be represented as lexical tokens. However, the main difficulty which may arise is the decision about which entity should be treated as an operand and which as an operator. There are other situations which also cause confusion. For example, a function reference may serve as an operand and operator at the same time.

Language Independency
Halstead's metrics n (program vocabulary), N (program length) and V (the program volume) are obviously language independent according to the definition given. According to the more usual definition of language independency they are language dependent. This is because these metrics are directly depend on counting the number of operators, operands, and their occurrence in the program, and the definition of operators and operands varies from language to another.

Modularity
Modularity may cause a reduction in a program's observed length, and therefore Halstead's metrics involving this will be affected by the modularization issue. According to modularity principles, the two similar parts must be combined in one subunit which must be isolated and then interfaced with the other subunits of the program. This modularity principle will cause a reduction in the observed program length, hence the value of Halstead's metrics will be decreased. Therefore, Halstead's metrics do reflect the reduction in the code which occurs as a result of applying a modularity technique. This is because the number of tokens will be decreased in the case of modular programming.

ABSTRACT METRICS
McCabe's metric [7] will be considered as examples for a comparison.

Applicability
McCabe's and the knot metric are designed to measure the control flow complexity of a program. Therefore, they are applicable to the implementation software development life cycle. The phase of the output of the design phase may be the following: graphs, tables, data design, program design, module design, etc. McCabe's metric can be applied to some of the components of the output of the

design phase, where the data flow graph and control flow graph are involved. For example, the design of each module, may be documented by a module input/output diagram, a module control flow diagram, a module data flow graph, etc., therefore it is possible to apply McCabe's metric to measure the complexity of the module control flow diagram. The result is a certain McCabe number, which has an indication about the design phase.

Language Independency
McCabe's metric is language independent, since it is based on the flow graph representation of any program.

Modularity
McCabe has suggested a value of "10" as the maximum complexity measure for a module to be manageable. Further, McCabe represented the control flow graph of a program as a directed graph. This directed graph can be reduced by replacing the proper sub graphs and then applying McCabe's metric, the obtained measure is the essential cyclomatic number. McCabe's essential cyclomatic complexity number can be used to discover whether a certain program needs further modularization or not. This is can be done by applying the essential cyclomatic number technique to a module in a program. If that module is highly complex, and its cyclomatic number could not be reduced to less than or equal "10", then a further modularization would require the redesign of that module. This shows that McCabe's metric in such case can be affected by the modularity criterion. That McCabe's metric may fail to reflect the modification which is caused by modularisation.

STRUCTURED METRICS
In this portion we apply the generated criteria of goodness of to Henry et al metrics [10].

Applicability
Henry et al structured metrics are based on the measurement of information flow between system components. Certain metrics are developed to measure procedure complexity, module complexity and module coupling. The procedure complexity metric is based on counting lines of code in a given procedure and the information flow connection of a procedure to its environment. The module complexity metric is based on the sum of the complexities of the procedures within the module. The module coupling metric is based on the extent to which two modules are coupled to each other. The code length metric can be applied to an implementation phase of the software system life cycle. This is because the code length metric was defined by Henry et al [10] as "the number of lines of text in the Source code for the procedure". This is equivalent to counting the number of statements in a program. However, such a metric cannot represent the full complexity of a procedure. This is because a procedure with small length but which contains predicate nesting and iterating statements is more complex than a procedure with a large length which contains simply a sequence of assignment statements. The coupling metric can be used as a tool during the implementation phase to indicate the effect of modifying a module on the other modules of a software system.

Language Independency
Henry et aI metrics deal software system connectivity by directly with the observing the flow of information among software system components. Since such information flow can be determined for any language therefore Henry et al metrics are language independent.

Modularity
It is one of the goals of modularisation to ensure that each procedure occurs in one and only one module [19]. When a procedure is located in more than one module, the modularisation becomes improper; this is because coupling will increase between the modules. The above problem can be discovered by Henry et aI module metric. This is because those procedures which violate the modularity principle will increase the value of the module metric and should be more prone to errors due to their connections to more than one module. To have a minimum value of the module metric is considered by Henry et al as a satisfactory result towards modularity. However, this is not the only way to minimise their metrics.

IV. CONCLUSION

As mentioned, the software metrics can be divided into three categories. That is primitive, abstract and structured. In the case of Halstead's metrics most of the relations between the parameters are nondeterministic and dependent on estimation. Further no attempt has been made to use operational research or probabilistic models for the relationships. Even though the existence of a high correlation between the measurements of program length, volume, size and the number of bugs in the program can be demonstrated, this does not ensure that program length, size and volume are essentially good predictors of errors. Neither can it be suggested that errors can be reduced by reducing the program length, size and volume [19]. The second category of software metrics (those which are based on the graph theory) are related to the order in which the various statements of a program are executed. Any alteration in the program statements sequential flow can be used as a measure of control flow complexity. Some of the control flow metrics are related to the important criteria such as the number of errors present in a piece of code and the time available to find and correct these errors [7]. Static measures have been created, in terms of which the source programs are analysed, and the software metrics are obtained and quantified. The researchers have also attempted to create a dynamic measure by introducing data flow, data structure, and analysing the program performance during execution. The final category of software metrics has certain advantages for a manager's overall understanding of system complexity and its impact on system costs and performance. Although the last category is based on structure properties of software design and seems to have some strong attractions, there is not sufficient research to give a true assessment of its value. Generally it can be concluded that, no approach at present can be considered as a standard and true measure of software systems. The metrics which are available now are not sensitive to errors. Metricians never show the negative results of their metrics whereas positive ones are published proudly.

REFERENCES

[1]. Ho~cker H., Itzfeldt M. and Timm M., "Comparative Description of Software Quality Measurement", GMD-Studien, March, 1984.
[2]. McCall J.A., Richard P.R. and Walters G.F., "Factors in Software Quality", Technical Report 77 CIS02, Vol. 1, 2, and 3, Sunnyvale, CA, General Electric Command and Information Systems, 1977.
[3]. Kitchenham B.A. and Walker J.G., "Test Specification and Quality Management, the Meaning of Quality", Deliverable A2l, Alvey Project SE/031, Report to the Alvey Directorate, 27th, May, 1986.
[4]. Jones T.C., "Measuring programming Quality and Productivity", I.B.M. System Journal, Vol. 17 No.1, PP. 9-64, 1978.
[5]. Yourdon E., "Structure Design", Fundamentals of a Discipline of Computer Program and System Design", 1978.
[6]. Boehom B.W., et al., "Characteristics of Software Quality", TRW Series on Software Technology, Vol. 1 North-Holland, 1978.
[7]. McCabe T.J., "A Complexity Measure ", IEEE Transaction on Software Engineering, SE-2, PP. 308-320, 1976',
[8]. Woodward M.R., Hennel M.A. and Hedly D., "A Measure of ControI FIow Complexity in Program Text", IEEE Transaction on Software Engineering PP. 45-50, 1979.
[9]. Henry S.M. and Kafura D., "Structure Metrics Based on Information Flow", IEEE Transaction on Software Engineering, Vol. SE-7, No.5, PP.510-S18, Sep. 1981.
[10]. Haney F.M., "Module Connection Analysis", A Tool for Scheduling Software Debugging Activities, Proceeding Fall Joint Computer Conference, PP. 173-179, 1972.
[11]. Albrecht A.J., "Measuring Application Development Productivity" in Proc. IBM Application Develop. Symposium, Monterey, CA, 14-17th Oct.1979. PP. 83
[12]. Halstead M.H., "Elements of Software Science", New-York, Elsevier North-Holland INC, 1977.
[13]. Kafura D. and Canning J., "A Validation of Software Metrics Using Many Metrics and Two Resources", Proceding of 8th International Conference on Software Engineering, August, PP. 378-385, 1985
[14]. Myers G.J., "Reliable Software Through Composite Design", N.V. Van-Nostrand, R. Heinhold 1975.
[15]. Myers G.J., "A Software Reliability, Principle and Practices", A WileY-Interscience publication, John Wiley and Sons, 1976.
[16]. Baker A.L. and Zweben S.H.,"A Comparison of Measures of Control Flow Complexity" IEEE Transaction on Software Engineering, Vol. SE-6, no.6 ,Nov. 1980, PP. 506 -512.
[17]. Baker A.C. and Zweben S.H." "The Use of Software Science in Evaluating Modularity", Concepts", IEEE Transaction on Software Enginnering Vol. SE-5, No.2 March 1979, PP. 110-120.
[18]. Kearney J.K., Sedlmeyer R.L., Thompson W.B., Gray M.A. and Adler M.A. "Software Complexity Measurement" Communication of ACM Vol. 29, No. 11, November, 1986.