CHAPTER - 2

SURVEY AND DESCRIPTION OF SOFTWARE METRICS
CHAPTER 2

SURVEY AND DESCRIPTION OF SOFTWARE METRICS

2.1 INTRODUCTION

Without software, computer hardware is of no use. A major share of the computing budget is incurred on software development or its purchase by each organization. So, software systems are very precious and important products for both software developers and users. Software is not a single attribute product but it has many characteristics which one can measure. For example, the size in lines of code, the cost of development and maintenance in rupees, the time for development in person-months, the size of memory required in bytes, and so on. Still, it is quite obvious that different observers of the same computer program may get different results, even when the same characteristic is measured [3,34].

For example, consider the lines of code property of a computer program. One observer may count all the lines present in the program including blank lines and comments. Another observer may drop comments and blank lines from the count by realizing that these do not affect the performance of the program. Therefore, a standard and precise definition of the line of code metric is required so that for the same program, different persons may get identical counts. Only under such standard, identical and homogenous conditions, we can compare the results of empirical studies conducted by different people at different times or places [3,35].

From time-to-time different software metrics have been developed to quantify various attributes of a software product. Broadly speaking, these may be grouped into two categories. These are

(i) Product metrics
(ii) Process metrics

These may be further classified as given in Fig. 2.1.

The software metrics, like size, which can be derived from the software itself are called product metrics. While, all those measurements of a software product which depend upon the development environment are called process metrics. Such metrics do not require the analysis of the program itself and are related to the development process. For example, measurement of the time required by a programmer to design, code and test a program is a process metric. This metric depends upon many things including the complexity of the problem, the knowledge and ability of the developer, the type of algorithm used, and the availability of the computer time during the development process etc. Therefore, if one computer program is to be developed by different programmers under different conditions, then their development time (for the same program) cannot be identical. Such metrics, like development time and effort estimation, may not be reproducible exactly. But, it does not mean that these are not useful and informative. In fact, these are the most important, useful, informative and interesting software metrics [3].

Here, we describe briefly some of the software metrics which seem to have received more attention of the researchers than others and have been studied in the present work.
Fig 2.1: Classification of Software Metrics
2.2 SIZE METRICS

For solving different problems on computer, programs are developed, written and implemented by different programmers. For achieving different objectives, programs are written in different programming languages. Some programs are written in C, few in Pascal and FORTRAN, some in COBOL, while others in C++, Ada languages and so on. Some programs are of good quality, well documented and written with latest software engineering techniques. While others are written in a “quick-and-dirty” way with no comments and planning at all. Despite all these, there is one common feature which all programs share - all have size.

Size measure is very simple, and important metric for software industry. It has many useful characteristics like:

- It is very easy to calculate once the program is completed.
- It plays an important role and is one of the most important parameter for many software development models like cost and effort estimation.
- Productivity is also expressed in terms of size measure.
- Memory requirements can also be decided on the basis of size measure.

The principal size measures which have got more attention than others are:

1. Lines of Code (LOC)
2. Token count
3. Function count

Lines of code

It is one of the earliest and the simplest metric for calculating the size of a computer program. It is generally used in calculating and comparing the productivity of programmers. Productivity is measured as LOC / man-month. Among researchers, there is no general agreement what makes a line of code. Due to lack of standard and precise definition of LOC measure, different counts may be obtained by different workers for the same program. Further, it also gives an equal weightage to each line of code. But, in fact some statements of a program are more difficult to code and comprehend than others. Despite all this, this metric still continues to be popular and useful in software industry because of its simplicity.

The most important characteristic of this metric is its precise and standard definition. There is a general agreement among researchers that this measure should not include comment and blank lines because these are used only for internal documentation of the program. Their presence or absence does not affect the functionality, efficiency of the program. Some observers are also of the view that only executable statements should include in the count, because these only support the functions of the program.
The predominant definition of LOC measure used today by various software personnel is:

"Any line of program text excluding comment or blank lines, regardless of the number of statements or parts of statements on the line, is considered a line of code (LOC). It excludes all lines containing program headers, declarations, and non-executable statements and includes only executable statements."

**Token Count**

The drawback in LOC size measure of treating all lines alike, can be solved by giving more weight to those lines which are difficult to code and have more "stuff". One natural solution to this problem may be to count the basic symbols used in a line instead of lines themselves. These basic symbols are called "tokens". Such a scheme was used by Halstead in his theory of software science [36]. In this theory, a computer program is considered to be a collection of tokens, which may be classified as either operators or operands. All software science metrics can be defined in terms of these basic symbols. The basic measures are:

- \( n_1 = \text{count of unique operators} \)
- \( n_2 = \text{count of unique operands} \)
- \( N_1 = \text{count of total occurrences of operators} \)
- \( N_2 = \text{count of total occurrences of operands} \)

An operator can be defined as a symbol or keyword which specifies an action. Operators consist of arithmetic, relational symbols, punctuation marks, special symbols (like braces, := ), reserved-word/keywords (like WHILE, DO, READ ) and function names like printf(), scanf() etc. A token which receives the action and is used to represent the data is called an operand. Operands include variables, constants and even labels.

In terms of the total tokens used, the size of the program can be expressed as:

\[ N = N_1 + N_2 \]

At present, there is no general agreement among researchers on counting rules for the classification of these tokens. These rules are made by the programmer for his/her convenience. The counting rules also depend upon the programming language [3,14,34].

**Function Count**

The size of a large software product can be estimated in a better way through a larger unit - called module than the LOC measure. A module can be defined as segment of code which may be compiled independently. For large software systems, it is easier to predict the number of modules than the lines of code. For example, let a software product require \( n \) modules. It is generally agreed that size of the module should be about 50 - 60 lines of code. Therefore size estimate of this software product is about \( n \times 60 \) lines of code. But this metric requires precise and strict rules for
dividing a program into modules. Due to the absence of these rules, this metric may not be so useful.

A module may consist of one or more functions. In a program, a function may be defined as a group of executable statements which performs a definite task. The number of lines of code for a function should not be very large. It is because human memory is limited and a programmer cannot perform a task efficiently if the information to be manipulated is large [37].

\[ \text{2.3 SOFTWARE SCIENCE METRICS} \]

Researchers generally agree that simple size measures like lines of code (LOC) are not adequate for determining software complexity and development effort. They are of the view that for this purpose a programming process model is needed. This model should be based upon manageable number of factors which affect the complexity and quality of the software systems. A number of researchers including Halstead [36], McCabe [38] have attempted to define such programming models.

Halstead’s model also known as theory of software science, is based on the hypothesis that program construction involves a process of mental manipulation of the unique operators \( n_1 \) and unique operands \( n_2 \). It means that a program of \( N_1 \) operators and \( N_2 \) operands is constructed by selecting from \( n_1 \) unique operators and \( n_2 \) unique operands. By using this model, Halstead derived a number of equations related to programming such as program level, the implementation effort, language level and so on. Also, it is one of the most widely studied theories and has been supported by a number of empirical studies [12,39-41].

An important and interesting characteristic of this model is that a program can be analysed for various features like size, effort etc. by simply counting its basic parameters \( n_1, n_2, N_1 \) and \( N_2 \) (defined in the previous section).

Program vocabulary is defined as

\[ n = n_1 + n_2 \]  

(1)

and program actual length as

\[ N = N_1 + N_2 \]  

(2)

One of the hypothesis of this theory is that the length of a well-structured program is a function of \( n_1 \) and \( n_2 \) only. This relationship is known as length prediction equation and is defined as

\[ N_h = n_1 \log_2 n_1 + n_2 \log_2 n_2 \]  

(3)

This length equation estimates the size of the program from the counts of unique operators \( n_1 \) and unique operands \( n_2 \). If the actual length \( N \) agrees well with the estimated value \( N_h \), then the program is considered as well structured.

Besides Halstead’s length equation (3), the following other length estimators have been suggested by some other researchers:
Jensen’s Program Length Estimator \[ N_f \]

It is described as:

\[ N_f = \log_2 (n_i!) + \log_2 (n_2!) \quad (4) \]

It was applied and validated by Jensen and Vairavan [42] for real-time application programs written in Pascal and found even more accurate results than Halstead’s estimator.

Zipf’s Program Length Estimator \[ N_z \]

It was defined by Zipf [14] and is given as:

\[ N_z = n [0.5772 + \ln(n) ] \quad (5) \]

where

\( n \) is program vocabulary given as:

\[ n = n_1 + n_2 \]

where

\( n_1 \): Number of unique operators which include basic operators, keywords/reserve-words and functions/procedures.

\( n_2 \): Number of unique operands.

Bimlesh’s Program Length Estimator \[ N_b \]

Another length equation similar to Halstead’s equation reported at the Annual Convention of Computer Society of India, 1986 [43] is given below:

\[ N_b = n_1 \log_2 (n_2) + n_2 \log_2 (n_1) \quad (6) \]

Where \( n_1 \) and \( n_2 \) are same as described in Halstead’s equation.

By using the four basic parameters \( n_1, n_2, N_1 \) and \( N_2 \), the theory of software science [36] further defines the following additional software metrics.

Program Volume \( (V) \)

The programming vocabulary \( n = n_1 + n_2 \) (set of unique operators and operands) used in writing a program, leads to another size measure which may be defined as

\[ V = N \log_2 n \quad (7) \]
It is called as the volume of the program. It may be interpreted as the number of mental comparisons needed to write a program of length $N$. The unit of measurement for program volume, $V$, is binary digit i.e. bit. Thus, total number of bits required to represent a program of length $N$ would be $N \cdot \log_2 n$.

It is assumed that during programming process, the human mind follows binary search technique in selecting the next token from the vocabulary of size $n$.

**Potential Volume ($V^*$)**

It is clear that an algorithm can be implemented through many different but equivalent programs. Out of these programs, one which has the minimum size is said to have the potential volume ($V^*$). It may be defined as:

$$V^* = (n_1^* + n_2^*) \log_2 (n_1^* + n_2^*)$$

where

$n_1^*$ is the minimum number of operators and $n_2^*$ is the minimum number of operands for an implementation.

Minimum number of operators, $n_1^*$, for any procedure is two — the procedure name and a grouping symbol that separates the procedure name from its parameters.

So, for any procedure

$$V^* = (2 + n_2^*) \log_2 (2 + n_2^*)$$

The minimum number of operands, $n_2^*$, is the number of unique input and output parameters. For small, simple programs, it can be calculated easily but for large, complex programs like compiler, operating system, it is difficult to compute.

**Program Level ($L$)**

An algorithm may be implemented in many different but equivalent ways. But, it is the level of the program which makes two implementations different and is defined as

$$L = V^* / V$$

The maximum value for program level, $L$, is 1. A program with $L = 1$ is said to be written at the highest possible level (i.e. with minimum size). Because, it is very difficult to determine potential volume ($V^*$), so Halstead gave an alternate formula for program level as:

$$L = \left( \frac{2 \cdot n_2}{n_1} \right) / \left( \frac{N_2}{N_1} \right)$$
Programming Difficulty (D)

The difficulty involved in writing a program can be defined as the inverse of the program level (L) and can be expressed as:

\[ D = \frac{1}{L} \] (12)

For a program, as the volume V increases, the program level L decreases and the difficulty D increases.

Programming Effort (E)

As already stated, the total number of mental comparisons required for writing a program of length N is \( N \log_2 n \). Further, for one mental comparison, the human mind has to perform a number of elementary mental discriminations (e.m.d.). Therefore, the effort required to implement a computer program increases as the size of the program increases. The programming effort E measure can be defined as:

\[ E = \frac{V}{L} \] (13)

The unit of measurement of E is e.m.d. (Elementary Mental Discriminations).

It is clear that more effort is required to implement a program at a lower level (higher difficulty) than another equivalent implementation at a higher level (lower difficulty).

Programming Time (T)

According to John Stroud [3,36], human mind can make a limited number of elementary mental discriminations per second. Halstead called this number as Stroud number and let it is denoted as B. Stroud claimed that the value of B ranges between 5 and 20. By using this number, time taken by a programmer to complete a programming task can be estimated in seconds as:

\[ T = \frac{E}{B} \] (14)

Generally, the value of B is taken as 18, since this number has given the best results for Halstead's earlier experiments.

Language Level (L)

At present, a number of programming languages are being used. Therefore, a software measure is required that expresses the power of the language. For this purpose, another metric, called as language level is defined by Halstead as:

\[ L = L \, V^* = L^2 \, V \] (15)

This metric is based on the hypothesis that for a given programming language, as \( V^* \) increases, L decreases in such a way that \( LV^* \) remains constant.
For the validation of Halstead's software metrics, it is necessary that programs should be devoid of any kind of impurity or redundancy. Impurities in an implementation may be incorporated due to poorly designed, poorly structured, lengthy and complex module. In a program, the following types of impurities may be included [15,36]:

- Complementary Operations
- Ambiguous Operands
- Redundant Conditions
- Dead Code
- Synonymous Operands
- Common Subexpressions
- Unwarranted Assignment, and
- Unfactored Expressions.

It is important that a program should not be ‘just’ a working program, but it must be a good quality program. Earlier, we have explained various characteristics of a good quality program and devoid of impurities is another step for it.

Though, the theory of software science metrics is very useful for the quantification of various aspects of a program. These metrics have, also, been supported by many experimental studies [15,22,33]. But, it reflects only one type of program complexity — size. It does not take into account the structure properties of the program or the modular interactions of the program. Therefore, it cannot be used to measure the overall complexity of a program [16,44].

2.4 CONTROL FLOW METRICS

There is general agreement among researchers that software modules with complex structures (with high density of branch instructions and loops) are more difficult to understand, debug and test than with simpler structures. Also, there is more possibility of errors in the maintenance phase for harder programs than simpler ones. For the quantification of complexity of a program, decision constructs play an important role. From time-to-time, several software metrics have been developed for the quantification of this particular feature of software modules. Some of these are briefly discussed here.

McCabe’s Cyclomatic Complexity Metric

The basis of this measure is the control flow graph of a program [38]. Due to its simplicity, it is one of the most useful and accepted metric. McCabe interprets a computer program as a set of strongly connected directed graph. Nodes represent parts of the source code having no branches and arcs represent possible control flow transfers during program execution. The notion of program graph has been used for this measure and it is used to measure and control the number of paths through a program. The complexity of a computer program can be correlated with the topological complexity of a graph [38, 45-47].

McCabe proposed the cyclomatic number, \( V(G) \) of graph theory as an indicator of software complexity. The cyclomatic number is equal to the number of linearly independent paths through a
program in its graph representation. For a program control graph G, the cyclomatic number, V(G), is given as:

\[ V(G) = E - N + P \]

where

- \( E \): The number of edges in graph G,
- \( N \): The number of nodes in graph G, and
- \( P \): The number of connected components in graph G.

For a strongly connected graph G, the value of \( P = 1 \). A graph G is said to be strongly connected if its each node is reachable from the every other node.

It can be easily shown that the cyclomatic complexity measure, V(G), can be calculated by the number of control structures (like if..then..else, while..do etc.) plus 1. Due to its ease for computation, it is one of the simplest, and the most widely accepted measure. Besides its simplicity, this measure has several other important features as described below:

- It is independent of program size.
- It depends only on decisions.
  - \( V(G) \) \( \geq 1 \).
  - \( V(G) = 1 \) for a linear sequence of code of any length.
  - It is equal to the maximum number of linearly independent paths in graph G.
  - \( V(G) \) is unaffected by adding or deleting non-branching statements.

The cyclomatic number, V(G), can also be computed just by knowing the number of decision structures involved in the source code. It is intuitive that number of errors are correlated to the number of control structures included in a software module. So, this measure may also be used for the estimation of errors (faults) in a computer program.

As stated earlier, due to its simplicity, McCabe’s V(G) is one of the most widely accepted complexity measure. However, it is not entirely convincing as a program complexity measure. One of its important shortcomings is the failure to account for the complexity due to unconditional GOTO’s. Because no decision is involved, so these are not counted. It, also, does not take into account the complexity due to linear sequence of statements and the nesting of control structures. So, several researchers [46-50] have suggested modifications to the use of this measure.

McCabe’s measure, V(G), depends upon the flow of control only and is insensitive to other parameters of program complexity such as size. McCabe observed that if the value of V(G) for a module becomes more than 10, then that module is more likely to be unreliable. So, such a module should be broken further to keep its V(G) \( \leq 10 \). This upper bound for V(G) for controlling the complexity of module seems reasonable, but it needs further empirical verification.

**Stetter’s Program Complexity Measure**

Halstead’s programming effort (E) does not differentiate among various kinds of control structures for the complexity which they contribute. For example, a loop structure is certainly more complex than an if..then..else clause. McCabe’s cyclomatic complexity measure, also, does not take into
account the effect of data i.e. constants, variables and operators. Baker [51] has shown that the
drawbacks of cyclomatic number and programming effort measures can be eliminated in a
synthesized program complexity measure. Following this idea, Stetter defined a synthesized
measure of program complexity which takes both components namely operations (statements) and
data (operators and operands) of a program into account. Both of these components of a program
contribute to the program complexity [52]. Thus, Stetter's metric accounts for the data flow
alongwith the control flow of the program which can be calculated from the source code. So, it
may be viewed as an extension of cyclomatic complexity metric. Stetter views the program as a
sequence of declarations and statements. It is given as:

\[ P = (d_1, d_2, \ldots, d_k, s_1, s_2, \ldots, s_m) \] (16)

where

- \( d \)'s are declarations,
- \( s \)'s are statements, and
- \( P \) is a program

Here, the notion of program graph [38] has been extended to the notion of flowgraph. A flowgraph
of a program \( P \) can be defined as a set of nodes and a set of edges. A node represents a declaration
or a statement while an edge represents one of the following:

i. Flow of control from one statement node say \( s_i \) to another \( s_j \).

ii. Control flow from a statement node \( s_i \) to declaration node \( d_j \) via a write access of a variable in
    \( s_i \) which is declared in \( d_j \).

iii. Flow from a declaration node \( d_j \) to a statement node \( s_i \) through a read access of a variable or a
    constant in \( s_i \) which is declared in \( d_j \).

This measure [52] is defined as:

\[ F(P) = e - n + n_e + n_t \] (17)

where

- \( n_e \): number of entry nodes, and
- \( n_t \): number of exit nodes.

Also, we have

\[ e = n + \sum_i (b(s_i) - 1) + \sum_j (a(d_j) - 1) \] (18)

where

- \( b(s_i) \): number of branches originating at a statement node \( s_i \),
- \( a(d_j) \): number of branches originating at a declaration node to a statement node or from a
  statement node to a declaration node.
But, a branch from a declaration node may be a read access and the one from statement to declaration node indicates a possible write access. So,

\[ a(d_j) = (\text{Read} + \text{Write}) \] accesses of the variable declared in \(d_j\).

Putting (18) in (17), we get

\[ F(P) = \sum_i (b(s_i) - 1) + \sum_j (a(d_j) - 1) + n_a + n_t \quad (19) \]

Since for a node from which only one edge originates, the term \((b(s_i) - 1)\) vanishes, so for a linear sequence of code, we have

\[ F(P) = \sum_j (a(d_j) - 1) + n_a + n_t \quad (20) \]

This complexity metric is an extension of McCabe's metric. Stetter's measure is an improvement over McCabe's measure because it does not give the same value for two different linear codes as in McCabe's metric. It also reflects that loop structure is more complex than an alternative construct.

2.5 INFORMATION FLOW METRICS

Control flow metrics do not take into account the complexity contributed by the interaction among modules. But, the information flow metrics deal with this type of complexity by observing the flow of information among system components or modules. It takes into account the connectivity among modules. Not much attempt has been made so far for constructing such kind of metrics. The metric given by Henry and Kafura [53] lies in this category.

**Henry and Kafura's Metric**

Henry and Kafura's complexity measure is based on the measurement of the information flow among system modules. It is sensitive to the complexity due to interconnection among system components. This measure includes complexities of modules within the system and also due to interfaces between the various components of the system [53].

The basis of this metric is that the complexity of a software module is defined to be the sum of complexities of the procedures included in the module. A procedure contributes complexity due to the following two factors:

i. The complexity of the procedure code itself

ii. The complexity due to procedure's connections to its environment.

The effect of first factor has been included through LOC (Lines of Code) measure. For the quantification of second factor, Henry and Kafura have defined two terms, namely, FAN-IN and FAN-OUT.
FAN-IN of a procedure is the number of local flows into that procedure plus the number of data structures from which this procedure retrieves information. While, FAN-OUT is the number of local flows from that procedure plus the number of data structures which that procedure updates.

So, the Henry and Kafura’s measure for procedure complexity is defined as:

\[
\text{Procedure Complexity} = \text{Length} \ast (\text{FAN-IN} \ast \text{FAN-OUT})^2.
\]

where length is taken as LOC and the term FAN-IN \ast FAN-OUT represents the total number of input - output combinations for the procedure.

The principal advantage of this metric over McCabe’s cyclomatic number is that it takes into account data-driven programs. The elements of this metric can be applied at design stage. So this metric is available at design stage - an early phase in software development life cycle. Thus, it may be used as a guide for restructuring of software with low cost.

Henry and Kafura validated it on a large software system - UNIX operating system and established that it has potential for identifying faulty components of large software systems. So, this metric is more useful for the quantification of complexity of large software systems having several components and it has potential for a new dimension of complexity (due to interaction of modules) as compared to other measures.

2.6 SUMMARY

In this chapter, we have conducted a survey of various important software complexity metrics. Through these metrics, the important parameters of a program like size, control structures and information flow among modules, have been reflected. These features contribute significantly to program complexity.

The two measures of complexity — software science and McCabe’s metric, have been explained at length. The first metric takes into account the complexity due to size only while the latter one considers it only due to control structures. Size metrics including lines of code (LOC), token count and function count have also been described.

Stetter’s program control flow, and Henry and Kafura’s information flow complexity metrics have also been explained here. Henry and Kafura’s metric is useful for the quantification of large software systems.

All aspects of complexity of a software module cannot be quantified by any single metric. Stetter’s measure is an attempt in this direction which combines two program characteristics: data flow (flow of operators and operands) and control flow. We have, also, attempted to design a metric which combines various aspects of complexity in Chapter 4.