Taxonomy Dimensions of Complexity Metrics

Bouchaib Falah[1], Kenneth Magel[2]
[1]*Al Akhawayn University, Ifrane, Morocco,*
[2]*North Dakota State University, Fargo, ND, USA*
[1]B.Falah@aui.ma, [2]Kenneth.magel@ndsu.edu

Abstract

Over the last several years, software engineers have devoted a great effort to measuring the complexity of computer programs and many software metrics have been introduced. These metrics have been invented for the purpose of identifying and evaluating the characteristics of computer programs. But, most of them have been defined and then tested only in a limited environment. Scientists proposed a set of complexity metrics that address many principles of object oriented software production to enhance and improve software development and maintenance. The aim of this paper is to present taxonomy of complexity metrics that, separately, evaluate structural and dynamic characteristics of size, control flow, and data. While most invented metrics applied to only the method and class levels of complexity, our approach uses metrics on each of the three levels: class, method, and statement.

Keywords: Complexity Metrics; Software Testing, Effectiveness, Data flow, Data usage; Taxonomy; Cohesion.

1. Introduction

Measurement makes interesting characteristics of products more visible and understandable [1, 2]. Appropriate measurement can identify useful patterns present in the product being measured [3]. It makes aspects and products more visible and understandable to us, giving us a better understanding of relationships among activities and entities. Measurement is not only useful, but it is necessary. It is needed at least for assessing the status of our applications, projects, products, and systems. Measurement does not only help us to understand what is happening during the development and maintenance of our projects, but it also allows us to control the interaction between the components of our project and encourages us to improve our projects and products.

There are a multitude of computer program software metrics that have been developed since the pioneering work of Halstead [4]. There are also several taxonomies that have been used to describe these metrics.

Nowadays, software is expected to have an extended lifespan, which makes the evaluation of its complexity at the early stages critical in upcoming maintenance. Indeed, complexity is proportional to the evolution of software. Software metrics were introduced as tools that allow us to obtain an objective measurement of the complexity of software. Hence, enabling software engineering to assess and manage software complexity. Reducing software costs is one of the major concerns of software engineering which creates an increasing need for new methodologies and techniques to control those costs. Software complexity metrics can help us to do so. In this paper, we would provide taxonomy of complexity metrics that can be served in reducing software costs. These metrics are used on each of the three levels: class, method, and statement.

2. Related Work

Many metrics have been invented. Most of them have been defined and then tested only in a limited environment. The most commonly used metrics for software are the number of lines of source code LOC (a rough measure of size), and Cyclomatic complexity (a rough measure of control flow).

Halstead software science [4] metrics are other common object oriented metrics that are used in the coding phase. Maurice Halstead's approach relies on mathematical relationships among the number of variables. His metrics, or what are commonly referred to as 'software science' [4], were proposed as means of determining quantitative measures directly from the operators and operands in the program. Halstead metrics

are used during the development phase with the goal of assessing the code of the program. Halstead's metrics are at the statement level, although they can be aggregated to form method and class level metrics.
Chidamber and Kemerer [5] proposed a set of complexity metrics that address many principles of object oriented software production to enhance and improve software development and maintenance. However, their metrics applied to only the method and class levels of complexity. They were evaluated against a wide range of complexity metrics proposed by other software researchers and experienced object oriented software developers. When these metrics are evaluated, small experiments are done to determine whether or not the metrics are effective predictors of how much time would be required to perform some task, such as documentation or answering questions about the software. Results have been mixed. Nevertheless, industry has adopted these metrics and others because they are better than nothing.

In recent years, much attention has been directed toward reducing software cost. To this end, software engineers have attempted to find relationships between the characteristics of programs and the complexity of doing programming tasks or achieving desirable properties in the resulting product such as traceability or security. The aim has been to create measures of software complexity to guide efforts to reduce software costs.

Our work applies a comprehensive suite of complexity metrics that can solve the problem of maximizing the effectiveness of software testing

3. Software Complexity Metrics

This paper uses software complexity metrics for object-oriented applications. Metrics for code that is not object oriented are not discussed in this research paper.
A metric is a measurement. Any measurement can be a useful metric. There are several reasons to use metrics in measuring the complexity of software, for instance:

- Prediction: metrics form the basis of any method for predicting schedule, resource needs, performance or reliability.
- Evaluation: metrics form the basis of determining how well we have done.
- Targeting: metrics form the basis for deciding how much effort to assign to which part of a task.
- Prioritization: metrics can form the basis for deciding what to do next.

Several researchers have proposed a wide variety of software complexity metrics. Each metric examines only one characteristic of software. This characteristic is one of:

- Size: how large is the software.
- Control Flow: either how varied is the possible flow or how deeply nested is the possible flow or how long is the possible flow.
- Data Usage: either how many data items are defined in the software or how many data items are related or how many values an attribute's value depend upon.

3.1. Size Metrics

One of the basic measures of a system is its size. Measures of software size include length, functionality, and complexity.

The oldest and most widely used size metric is the lines of code. The lines of code are common object oriented metrics that are used in the coding phase. There are two major ways to count the lines of code depending on what we count: a physical line of code (LOC) and a logical line of code (LLOC). While the common definition of LOC is the count of lines in text of the program's source code including comment lines, LLOC is defined to be the number of statements.
For example: if we consider the following Java fragment code:

```
if (int i = 0; i < 4; i++)  x[i] = i + 1;     // this is a line of code example.
```

In this example: LOC = 1 and LLOC = 2.

Another common OO metrics that are used in the coding phase were provided by Halstead software science [4]. Halstead's approach is based on the assumption that a program should be viewed as an expression of language. Halstead believed that the complexities of languages are an essential part of the reasons a programmer might find complexity in the program code. Therefore, he bases his approach on the mathematical relationships among the number of variables, the complexity of the code and the type of programming language statements

Because our research is related to Object Oriented Java Application, we will adopt the Halstead metrics to calculate the number of operators that are contained in each statement of a Java code program, then we will extend this metric to compute the total and the maximum number of operators of all statements within each method, and furthermore, we will compute the total and the maximum number of operators in all methods within the class. That means that we will use the number of operators in all three levels: class, method, and statement.

3.2. Control Flow Metrics

Another object oriented metric that is used in coding phase is McCabe Cyclomatic metric [6, 7]. Thomas McCabe developed his complexity metric in 1976. His approach was based on the assumption that the complexity

of software is related to the number of control paths generated by the code [6]. In other words, the code complexity is determined based on the number of control paths created by the code. This means that, in order to compute a code complexity, the number of decisions (if/then/else) and control statements (do while, while, for) in the code are the sole criterion for this purpose and therefore must be determined. For example, a simple function with no conditionals has only one path; a function with two conditionals has two paths. This metric is based on the logic that programs with simple conditionals are more easy to understand and hence less complex. Those with multiple conditionals are harder to understand and hence, more difficult and complex.

The control flow graph, G, of any given program can be drawn. Each node of the graph G corresponds to a block of code and each arc corresponds to a branch of decision in the program. The McCabe cyclomatic metric- [8] of such graph can be defined as:

$CC(G) = E - N + 2P$ (1) where,

- E: is the number of edges of G.
- N: is the number of nodes of G.
- P: is the number of connected components.

The formula (1) can also be written as:

$CC(G) = D + 1$ (2) where,

- D: is the number of decisions inside of the code.

Even if this information supplies only a portion of the complex picture, McCabe [7] tried to extend his metric into an architectural design and developed a testing methodology that integrates the notion of design complexity with the testing requirement.

3.3. Data Metrics

Data complexity metrics car be divided in two different aspects: data flow and data usage. Data flow is the number of formal parameters of activities and the mappings between activities' data [9]. We will define Data usage for a statement to be the number of variable values used in that statement plus the number of variable assigned new values in that statement.

The development of test cases of many researchers was based on the program unit's variables. The emphasis of test cases was based on data and data flow or Data-Usage Path [10]. Chidamber and Kemerer metrics [5], also known as C&K metrics, were among the first family of related metrics that address many concerns of OO designers including relationships such as coupling, cohesion, inheritance, and class size [11]. The notion of cohesion and the various complexity metrics associated with the cohesion are also related to data variables. In OO, the most widely C&K metric used example, when cohesion is related to instance variables, is Lack of Cohesion in Methods (LOCM) [12, 13].

Chidamber and Kemerer proposed a set of metrics that cover not just the data aspect but also cover other different aspects.

The C&K metrics are computed for each class in an application. Most of the metrics are at the class level while a few are at the method level. Figure 1, for example, illustrates how the C&K metrics would be apportioned among taxonomy dimensions.

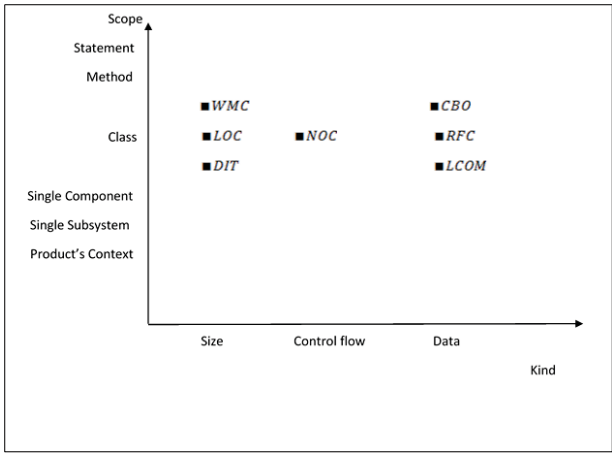

Figure 1. Taxonomy Dimensions of C&K Metrics.

While C&K metrics are used only at class and method levels, our approach uses metrics on each of the three levels: class, method, and statement.

Figure 2 illustrates how our suite of complexity metrics would be apportioned among our taxonomy dimensions.

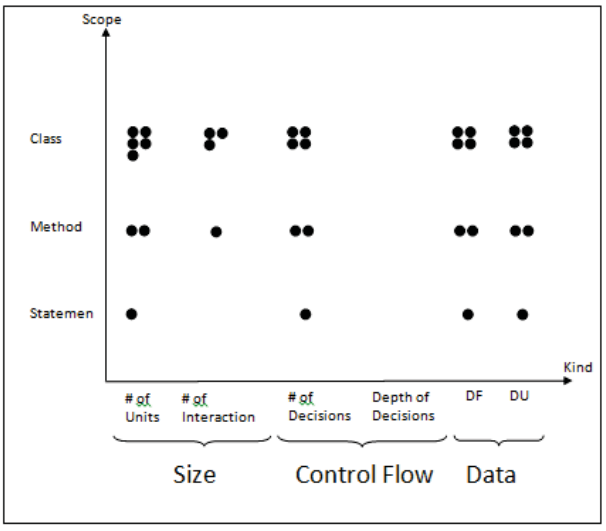

Figure 2. Taxonomy Dimensions of Our Approach.

4. Comprehensive Taxonomy of Metrics

Software engineers use measurement throughout the entire life cycle. Software developers measure the characteristics of software to get some sense of whether the requirements are consistent and complete, whether the design is of high quality, and whether the code is ready to be tested. Project Managers measure attributes of the product to be able to tell when the software will be ready for delivery and whether the budget will be exceeded. Customers measure aspects of the final product to determine if it meets the requirements and if its quality is sufficient. And maintainers must be able to assess and evaluate the product to see what should be upgraded and improved.

Software metrics usually are considered in one or two of four categories:

- Product: (e.g. lines of code)
- Process: (e.g. test cases produced)
- People: (e.g. inspections participated in)
- Value to the customer: (e.g. requirements completed)

In our work, we will concentrate on product metrics as selectors for test cases. Previous work using metrics almost always considered only a small set of metrics which measured only one or two aspects of product complexity.

Our work starts with the development of a comprehensive taxonomy of product metrics. We will base this taxonomy on two dimensions: (1) the level of the product to which the metric applies; and (2) the characteristic of product complexity that the metric measures.

In future work, we hope to produce a comprehensive taxonomy from the other kinds of metrics.
The scope of consideration dimension includes the following values:

(1) the product's context including other software and hardware with which the product interacts
(2) the entire product
(3) a single subsystem or layer
(4) a single component
(5) a class
(6) a method
(7) a statement

For the initial uses of this taxonomy reported in this paper, we will use only (5), (6), and (7) since they appear to be the most relevant scopes for unit testing. Future work may add (3) and (4) as we consider integration testing. Values (1) and (2) may be used for system testing.

The complexity kind dimension includes the following values:

1) Size
2) control flow
3) data

Each of these values in turn has sub-values.

For size, the sub-values are:

a) number of units (e.g. statements)
b) number of interactions (e.g. number of method calls)

For control flow, the sub-values are:

a) number of decisions
b) depth of decisions

For data, the sub-values are:

a) data usage
b) data flow

4.1. Metrics at Statement Level

4.1.1. Data Complexity. In our research, we consider two separate aspects, data flow and data usage. Data flow is based on the idea that changing the value of any variable will affect the values of the variables depending upon that variable's value. However, data usage is based on the number of data defined in the unit being considered or the number of data related to that unit. We will define data usage for a statement to be the number of variable values used in that statement plus the number of variable assigned new values in that statement.

Data flow complexity measures the structural complexity of the program. It measures the behavior of the data as it interacts with the program. It is a criteria that is based on the flow of data through the program. This criteria is developed to detect errors in data usage and concentrate on the interactions between variable definition and reference.

Several testers have chosen testing with data flow because data flow is closely related to Object Oriented cohesion [12, 14]. One measure of class cohesion is how methods are related through common data variables.

Data flow testing is a white box testing technique that can be used to detect inappropriate usage of data values due to coding errors [15]. For instance, a programmer might use a variable without defining it or might define a variable without initializing it (e.g. int a; if (a==1) {…}).

A program written in an OO language, such as Java, contains variables. Variables are defined by assigning values to them and are used in expressions. An assignment

statement such as: x = y + z ; defines the variable x. This statement also makes use of variables y and z. In this case, the variable x is called a definition while the variables y and z are called uses.

The declaration statement such as: int x, y, z; defines three variables x, y, and z. The three variables are assumed to be definitions.

In our research, data flow will be estimated for each statement in a method by counting how many active data values there are when the method executes. Active data values will be counted by determining which variable assignments could still be active when this statement begins execution plus the number of assignments done in this statement. As an example, let us consider the following Java class:

```
class DataFlowExample{
public void method1(int a, int b){
private int c = 0;
                                a = c + 3;
                                b = a + 1;
                                }
                        }
```

In the first statement of this code, the variable c is a definition. The same variable c is a use in the second statement. Thus, the data flow of this statement is 1.

In the second statement, a is a definition and c assigned a value. The variable a is a use in the third assignment. Thus the data flow value of the second statement is 2.

In the third statement, b is a definition and a assigns a new value. The variable b is no longer active before the method executes. Thus the data flow value of this third statement is 1.

On the other hand, as an example of data usage, let us consider the statement assignment: $x = y + z$.

The variables y and z are used, and the variable x is assigned a new value in the statement. Thus the data usage of this statement is 3.

4.1.2. Control Flow Complexity. In our research, we will use one control flow measure, the scope metric [16]. For each statement, we will count how many control constructs (do while, if-else, for, while …) contain this statement.

For example, assume that Figure 3 illustrates a statement fragment code of a return method named method C within the class "class C".

The construct level statements in this code are the statements numbered (6), (11), and (14).

```
1.      public class classC {
2.      public boolean methodC()
3.      {
4.              int x=0, y = 0, z = 0;
5.              boolean flag = false;
6.              while(flag)
7.              {
8.                      x = 2;
9.                      y = x + 1;
10.                     z = x + y;
11.                     if(y > 0)
12.                     {
13.                             y = y +1;
14.                             if(z == 0)
15.                                     z+= 2;
16.                             x+= 5;
17.                     }
18.             }
19.             return flag;
20.     }
21.     }
```

Figure 3. Java Code – Scope Metric Example.

Table 1 shows the scope metric value of each statement in the code of Figure 3.

Table 1. Scope Metric Values of Statements of Figure 5.

Statement	Construct Level contains the statement	Scope Metric Value
(4), (5)	None	0
(8), (9), (10)	(6)	1
(13)	(6), (11)	2
(15)	(6), (11), (14)	3
(16)	(6), (11)	2
(19)	None	0

4.1.3. Size Complexity. Our size metrics relied on the Halstead Software Science Definition. We will use a simplified version of Halstead's operators count discussed previously. Halstead's software science is one traditional code complexity measure that approaches the topic of code complexity from a unique perspective. Halstead counted traditional operators, such as + and ||, and punctuations, such as semicolon and ( ), where parentheses pair counted as just one single operator.

In our work, we will count just traditional operators for simplicity by counting the number of operators used in each statement of the code.

Figure 4 shows the metrics used in this research at the statement level. These four metrics will be used as roots to derive other complexity metrics that will be used at the method level and class level.

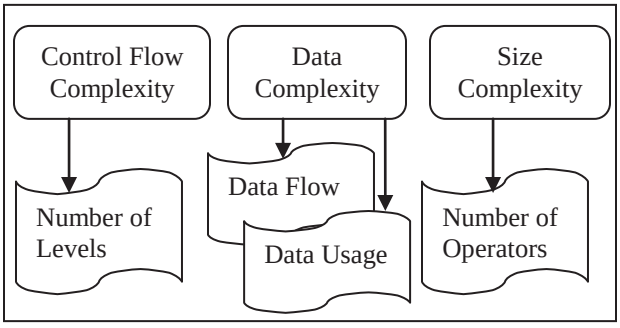

Figure 4. Complexity Perspectives and Metrics at Statement Level.

4.2. Metrics at Method Level

Since the method constitutes different statements, we will use both the sum of each metric for the statements within that method and the maximum value of each metric for the statements within that method. In addition to the sum and the maximum of these metrics, we will use another single level metric that counts the number of other methods within the same module (class) that call this method.

An example of this additional metric is shown in Figure 5.

```
public class ClassA {
        int[] arr = {1,2,3,5,6,9};
        public void method1()
        {
                for (int i=0; i<arr.length; i++)
                        System.out.println(arr[i]);
                method2(0);
        }
        public int method2(int y)
        {
                method1();
                for(int i=0; i<arr.length; i++)
                        y = y + arr[i];
                return y;
        }
        public void method3()
        {
                for(int i = 0; i< arr.length; i++)
                        arr[i] = arr[i] + method2(1);
                method1();
        }
}
```

Figure 5. Example of Other Methods that Call a Method within the Same Class.

For each method within the class "ClassA", the number of other methods that call that method with the same class is shown in Table 2.

Table 2. Metric Results of the Code in Figure 5.

Method	Other methods that call this method	Metric Value
method1	method2, method3	2
method2	method1, method3	2
method3	None	0

Figure 6 illustrates the nine metrics that will be used to measure the complexity of a method. Eight of these nine metrics are derived from the four metrics defined at statement level.

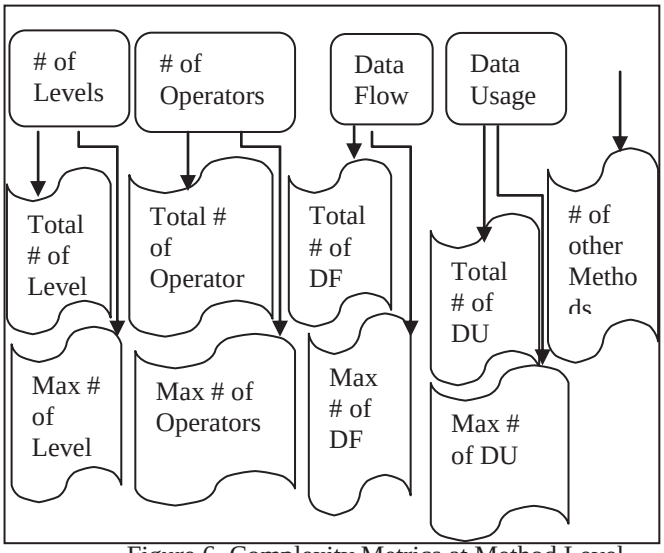

Figure 6. Complexity Metrics at Method Level.

4.3. Metrics at Class Level

At the class level, we will use both the sum of each metric for the methods within the class and the maximum value of each metric for the methods within the class. We will then add two additional metrics: the in-out degree of that class, which is the number of methods outside of that class that are called by at least one method in that class, and the number of public members within the class. The public members within a class are defined as the public fields and the public methods defined in that class.

As a summary of the comprehensive taxonomy of metrics that will be used in our research, for each executable statement within a method we will have 4 metrics that emerged from three complexity dimensions:

- Data Dimension: active data values and Data usage values.
- Control Dimension: scope metric.
- Size Dimension: number of operators.

For each method, we will have nine metrics. 2 metrics constitute the total and the max of the metrics of each statement within that method plus the number of other methods that call that method.
For each class, we will have twenty metrics, two metrics compose the total and the max of each of the 9 metrics that will be used for the method within that class, plus two more metrics including the number of methods outside of that class that are called by at least one method in that class, and the number of public members within the class.

5. Conclusion

This paper aims at developing a comprehensive taxonomy of product metrics that can be used to target test cases. This taxonomy is based on the metric dimension (product level) and the kind dimension (product complexity characteristic). We used the scope metric dimension values of class, method, and statement. We considered kind dimension values of size, control flow, and data. The three kind dimension values of product complexity have sub-categories. The size has the number of units and the number of interactions. The control flow has the number of decisions and the depth of decisions. The data has the data flow and the data usage.

In our work, we used at least one sub-category from each complexity kind dimension value. For the size, we used the number of units and the number of interactions. For the control flow, we used only number of decisions. For the data, we used data flow and data usage.

Another contribution of this research was the use of summation and maximum to build larger scope metrics from smaller scope metrics.

6. References

[1] B. Falah, K. Magel. "Test Case Selection Based on a Spectrum of Complexity Metrics". *Proceedings of 2012 on International Conference on Information Technology and Software Engineering (ITSE),* Lecture Notes in Electrical Engineering , Volume 212, 2013, pp. 223-235

[2] B. Falah, K. Magel, O. El Ariss. "A Complex Based Regression Test Selection Strategy", Computer Science & Engineering: *An International Journal (CSEIJ),* Vol.2, No.5, October 2012

[3] B. Falah. "*An Approach to Regression Test Selection Based on Complexity Metrics*" , Scholar's Press,
ISBN-10: 3639518683, ISBN-13: 978-3639518689, Pages: 136, October 28, 2013

[4] M.H. Halstead, "Elements of Software Science," Operating and programming systems series, New York: *Elsevier* North-Holland, 1977.

[5] S. R. Chidamber and C.F. Keremer, "A Metric Suite for Object Oriented Design," *IEEE Transactions on Software Engineering*, Vol. 20, No 6, June 1994, pages 476- 493

[6] T. J. McCabe and Charles Butler, "Design Complexity Measurement and Testing," *Communications of the ACM*, Vol. 32, Issue 12, December 1989.

[7] Thomas J. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, Vol. SE-2, No.4, December 1976.

[8] M. Clark, B. Salesky, C. Urmson, and D. Brenneman, "Measuring Software Complexity to Target Risky Modules in Autonomous Vehicle Systems," *AUVSI Unmanned Systems North America*, June 2008.

[9] J. Cardoso, "Control-Flow Complexity Measurement of Processes and Weyuker's Properties," *Word Academy of Science, Engineering and Technology,* August 2005.

[10] S. Rapps and E. Weyuker, "Selecting Test Data Using Data Flow Information," *IEEE Transactions on Software Engineering*, Vol. SE- 11, No. 4, April 1985, pp. 367-375.

[11] R. Harrison, S. J. Counsell, and R.V. Nithi, "An Investigation into the Applicability and Validity of Object Oriented Design Metrics," *Empirical Software Engineering*, Vol. 3, Issue 3, September 1998.

[12] S. R. Chidamber and C.F. Keremer, "A Metric Suite for Object Oriented Design," *IEEE Transactions on Software Engineering*, Vol. 20, No 6, June 1994, pages 476- 493

[13] S.R. Chidamber and C. F. Kemerer, "Towards A Metrics Suite for Object Oriented Design," *In Proceeding of the ACM Conference on Object-Oriented Programming Systems*, Languages, and Applications. Vol. 26, Issue 11, November 1991.

[14] F. Damereu, "A Technique for Computer Detection and Correction of Spelling Errors," *Communications of the ACM*, Vol. 7, Issue 3, March 1964.

[15] J.P. Myers, "The Complexity of Software Testing," *Software Engineering Journal*, January 1992, pp. 13 – 24.

[16] H. F. Li and W. K. Cheung, "An Empirical Study of Software Metrics," *IEEE Transactions on Software Engineering*, Vol. 13, Issue 6, June 1987.