A Survey on Metric of Software Complexity

Sheng Yu, Shijie Zhou
School of Computer Science and Engineering
University of Electronic Science and Technology of China
Chengdu, China
yusheng123@gmail.com, sjzhou@uestc.edu.cn

Abstract—With the evolution of the software development, the scale of the software is increasingly growing to the extent that we cannot hand it easily. Some metrics are proposed to measure the complexity of software in last a few years. This article aims at a comprehensive survey of the metric of software complexity. Some classic and efficient software complexity metrics, such as Lines of Codes (LOC), Halstead Complexity Metric (HCM) and Cyclomatic Complexity Metric (CCM), are discussed and analyzed first. Then, some other approaches driven from above classic metrics are also discussed. The comparison and the relationship of these metrics of software complexity are also presented.

Keywords- complexity measure, software complexity, software measuremen, project management.

I. INTRODUCTION

In the whole life cycle of the software, both the development and the maintenance are of high cost. If the complexity of software is decreased, the cost of software would also be decreased.

Early in 70th, the complexity of the software had attracted the public attentions. The modularization program style and the object-origin program style are both introduced to lower the complexity of the software. More or less, these approaches have moderated the software complexity. However, with the increasing size of software, the software complexity has beyond our control. For example, in the 60th the software for the Apollo Project has codes of 600,000 lines. But now the core products of Microsoft, Windows Vista, Microsoft Office 2007 and Exchange Server 2007, inconceivably approximate to codes of 200,000,000 lines.

Although some useful metrics are employed, the current solutions are not enough to settle down this rigorous problem. Both the computer researchers and the software engineers do not cease to find more powerful and effective metric of software complexity. So, a survey on the metrics of software complexity is necessary to understand and master the state of the art in this field. This paper aims to giving a comprehensive survey on the research and the development of the metrics of software complexity.

The rest of the paper is organized as follows: in section 2, the definition and the classification of software complexity metric are introduced. The classic software complexity metrics and their variations are analyzed in section 3. In section 4, some other related research works are discussed. In section 5, the trend and future work are discussed. Finally, we make the conclusion in section 6.

II. DEFINITIONS AND CLASSIFICATION OF THE SOFTWARE COMPLEXITY

2.1 The metric of software

The software metric is the measurement, usually using numerical ratings, to quantify some characteristics or attributes of a software entity. Typical measurements include the quality of the source codes, the development process and the accomplished applications. Some major measurements are listed in table 1.

<table>
<thead>
<tr>
<th>Role</th>
<th>Measurements</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td>Usability, Simplicity, Stability, Cost…</td>
</tr>
<tr>
<td>Designer</td>
<td>Extendibility, Scalability, Manageability…</td>
</tr>
<tr>
<td>Programmer</td>
<td>Complexity, Maintainability...</td>
</tr>
</tbody>
</table>

2.2 The metric of software complexity

The metric of the software complexity is an essential and critical part of the software metric. The metric of the software complexity focuses on the quality of source codes.

The complexity of software can be divided into three classes: the essential complexity, the selecting complexity and the incidental complexity [1]. The essential complexity is determined by the problems that the software tries to solve. The selecting complexity is determined by the program languages, the problem modeling methods and the software design methods. The incidental complexity is determined by the quality of the involved implementer. Some information about them is shown in the table 2.

<table>
<thead>
<tr>
<th>Class</th>
<th>Origin</th>
<th>Influence</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Essential</td>
<td>Problem</td>
<td>Huge</td>
<td>Extreme hard</td>
</tr>
<tr>
<td>Selecting</td>
<td>Method</td>
<td>Medium.</td>
<td>Hard</td>
</tr>
<tr>
<td>Incidental</td>
<td>Implement</td>
<td>Small</td>
<td>Fair</td>
</tr>
</tbody>
</table>

The target of measuring the software complexity is to manage and reduce the incidental complexity, and improve the development of the software and the software itself.

2.3 The classifications of the metrics of software complexity

Before discussing the details of the software complexity metrics, we classify the metrics in some ways.
On one hand, we can classify the metrics by the time when they are used in software lifetime. The classification, of 8 popular metrics sets, which are listed in the [2], is shown in the table 3. The book [3] discusses the detail of the software package metrics. The cohesion [4][5][6] is divided into seven grades. Coupling [7][8][9] has six grades. Coupling is usually contrasted with cohesion. The software with high cohesion and low coupling is high-quality.

On the other hand, we can classify the metrics of software complexity by what they are calculated on. The classification of the 8 metrics sets is shown in the table 4.

III. CLASSIC METRICS OF SOFTWARE COMPLEXITY AND THEIR VARIATIONS

During the 60th and 70th, three classic and important metrics, LOC, HCM and CCM, were invented. They are considered to measure the software in three different aspects, correspondingly, the length, the volume, and the structure.

With the development of the software design method, some metrics, which aim at special programming method, such as the Object-Origin programming and Aspect-Origin programming, have been introduced. But none of them is considered to measure the software in three different aspects, correspondingly, the length, the volume, and the structure.

And for their specialization, we do not discuss them here.

3.1 Lines of Code (LOC)

3.1.1 Definition and computing of LOC

The LOC is generally considered as the count of the lines in the source code of the software. Usually, the LOC only considers the executable sentence. Also sometimes, the LOC is estimated by the other factors [2]. LOC is independent of what program language is used. The LOC evaluates the complexity of the software via the physics length.

3.1.2 Some findings

The LOC is based upon two rules: the relationship between the count of code lines and the bug density, the independence between the bug density and the program language.

The early research [10] was done on some FORTRAN modules with LOC less than 200. The [10] discovered that the less LOC one module has, the higher bug density the module has. Then, some studies on the software written in Pascal, PL\S and assemble language prove that the bigger modules have higher bug density when the LOC equals to 500 or even higher [11]. The later work indicates that, there actually is a relationship of U type curve between the LOC and the bug density. When the LOC is low, the bug density is decreasing with the increase of the module size, LOC. But when the LOC is high, the bug density is increasing with the increase of the LOC.

On the other hand, the LOC ignores the structure of the software. For example, comparing with a longer code segment with many branches has a lower LOC. But the latter is much more complex than the former, and has a much higher rate to have some bugs.

Secondly, the LOC is independent of the program language. The effect of the LOC is also good. However, there are some shortages of the LOC.

First of all, the LOC metric ignores the different complexity of each code line. For example (all the examples in this article are using the C program language, and all the variable definitions are ignored), the code “i=(t+++func1(a,b))/func2(c,d)” have the same effect on the LOC. But based upon the basic professional knowledge and experience, the later is much more complex than the former, and has a much higher rate to have some bugs.

Secondly, the LOC ignores the structure of the software, such as branches and jumps. For example, comparing with a longer code segment without branches and jumps, one code segment with many branches has a lower LOC. But the latter is more complex from the view of programming.

Finally, the LOC doesn’t take the difference of the programmers’ abilities into account. The LOC is based on the relationship between the count of code lines and the bug density. The bug density has a strong connection with the programmer’s personal ability, habit and skill. So the LOC’s best range varies from one programmer to another.

3.2 Halstead Complexity Metric and its variations

3.2.1 Definition and computing of HCM

In 1977, Maurice Howard Halstead introduced the concept of software science. He began to use scientific
methods to analyze the characteristics and structure of the software. The idea resulted in the introduction of the HCM. The HCM is calculated on the count of the operators and operands [14]. The operators are symbols used in expressions to specify the manipulations to be performed. The operands are the basic logic unit to be operated. The HCM measures the logic volume of the software. Firstly, the HCM compute the following parameters:

\[
\begin{align*}
&n_1 = \text{the number of distinct operators} \\
&n_2 = \text{the number of distinct operands} \\
&N_1 = \text{the total number of operators} \\
&N_2 = \text{the total number of operands}
\end{align*}
\]

From these numbers, some indicators can be calculated:

- **Expected software length:** \(H = n_1 \log_2(n_1) + n_2 \log_2(n_2)\)
- **Software length:** \(N = N_1 + N_2\)
- **Software vocabulary:** \(n = n_1 + n_2\)
- **Volume:** \(V = N \times \log(n)\)
- **Level:** \(L = \frac{V^*}{N} = \frac{(2^{n_1} - 1)(n_2/N_2)}{n_1 + n_2} \)
- **Difficulty:** \(D = \frac{V}{V^*} = \frac{(n_2/2)}{(n_1/2)(N_2/n_2)}\)
- **Programming Effort:** \(E = V \times D\)
- **Error Estimate:** \(B = V/S^*\)
- **Programming Time:** \(T = E/18\)

The \(V^*\) is the software’s ideal volume. We commonly use the formula, \(V^* = (n_1 N_1 / 2 n_2) (N_1 + N_2) \log_2(n_1 + n_2)\), to estimate the \(V^*\). The \(S^*\) means the programmer’s ability. Halstead sets \(S^*\) for a fixed value of 3000.

### 3.2.2 Advantages and disadvantages of HCM

The HCM doesn’t need a deep analysis of the software’s logic structure. It is easy to calculate, has nothing to do with the program language, and can be used to forecast the bug density. Anyway, the HCM has some shortages.

The HCM only considers the complexity from the data stream, while ignores the complexity from the control flow. Using the HCM, the operators and operands of the codes, with or without some branches and jumps, are calculated with no difference. But it’s undoubted that, the codes with some branches and jumps are more complex.

What’s more, the value of \(S^*\) is too rigid. Different projects have different methods of project management. And project staff’s ability and experience vary from one to another. In so different projects, the use of the experiential and fixed value of \(S^*\) is doubtable.

### 3.2.3 HCM’s variations and some analysis of them

For overcoming these shortages, some altered methods are introduced. The article [15] introduced a Weighted HCM (WHCM) and a Data Stream and Control Stream Mode (DCM).

The WHCM gives different operators different weights. Actually, the complexity doesn’t come from the operators themselves, but the branches and jumps produced by the operators. But in the WHCM, the weight of the codes inside a branch is the same as that outside.

Furthermore, the WHCM calculate the complexity of a loop code segment by this, the HCM value of the codes inside the loop multiply the loop times. But how many times the codes run only can be determined when the software runs, so we can’t measure it statically and use it to estimate the complexity of the software. Besides, how many times the codes run is on the field of the algorithm complexity, rather than software complexity. How many times the codes run can’t influence the software’s logic complexity.

The WHCM also gives different operands different weights. Different data types do influence the operating efficiency. But their logic complexity is of no difference.

At the last, the WHCM takes the documents of the project into consideration. The WHCM uses the Capability Maturity Model for Software (SW-CMM) to measure the project’s documents and modify the HCM. The prime task of the CMM is monitoring and improving the project management, rather than the coding process. Therefore, the CMM shouldn’t be an indicator of the software complexity.

The DCM attempts to combine the data flow and control flow to measure the complexity. Its calculation formula is:

\[
DC = \frac{1}{\beta \lambda} \sum_{i=1}^{n} \left[ w_1 \times \frac{H(i)}{\text{Max}(H(j))} + w_2 \times \frac{M(i)}{\text{Max}(M(j))} \right]
\]

The \(n\) means that the codes have \(n\) modules. The \(H(i)\) is the HCM value of the module \(i\). The \(M(i)\) is the CCM value of the module \(i\). The \(w_1\) and \(w_2\) are the weights of the data flow and the control flow correspondingly. And \(w_1 + w_2 = 1\).

The \(\beta\) means the abilities of the developers. According to the levels of the CMM, the corresponding values of the \(\beta\) are 0.2, 0.4, 0.6, 0.8, and 1.0. The \(\lambda\) means the level of the development tools.

There are many limitations in the DCM. Firstly, the existence and value of the \(\beta\) and \(\lambda\) are questionable and maybe unsuitable. Because there is no universal standard to determine the merits and demerits of development tools, and different development tools have their own adaptive fields, the \(\lambda\) should not be used. Secondly, the DCM can’t analyze a module separately. When there is only one module, the formula becomes that, \(DC = 1/(\beta \times \lambda)\). At this point the complexity of the software has nothing to do with the software itself. Finally, when the complexities of the modules differ too dramatically, the DCM will be ineffective. For example, we have 10 modules, one of which (may set it the module 1) has a high complexity, \(H(1) = M(1) = 100\). The rests are simple modules, \(H(i) = M(i) = 1 (i = 2, 3, \ldots 10)\).

Now, using the formula, we get that \(DC = 1.09/(\beta \times \lambda)\). At this point, the complexity of the software has little to do with the software itself.

### 3.3 Cyclomatic Complexity Metric and its variation

#### 3.3.1 Definition and computing of CCM

In 1976, based upon the topological structure of the software, Thomas J. McCabe introduced a software complexity metric named McCabe Cyclomatic Complexity Metric [16]. In the following years, McCabe introduced many other metrics of the software complexity.

The CCM is the most basic and important metric among all. All the other McCabe metrics are calculated basing on the CCM. The CCM is a measurement of the complexity of a module's decision structure graph. Imagining that the exit node branches back to the entry node, the control flow graph is strongly connected then. The value of the CCM is the
number of linearly independent paths and therefore, the minimum count of paths that should be tested, because any path can be expressed as a linear combination of some linearly independent paths. Thus the McCabe CCM has a solid theoretical foundation of mathematics.

The nodes of the graph correspond to the code lines of the software, and a directed edge connects two nodes if the second node might be executed immediately after the first one. If the conditional evaluation expression is composite, the expression should be broken down. For example, the expression “if(c1 & & c2){}” should be treated as “if(c1){if(c2){}}”. The control flow graph of a module has one and only one entry node and exit node. If one control flow graph has e edges and n nodes, the CCM value of the corresponding module is \( V(G) = e - n + 2 \). The figure 1 is a flow graph has e edges and n nodes, the CCM value of the expression “if(c1 && c2){}” should be less than 10, and not be more than 20. When the conditional evaluation expression is composite, the second node might be executed immediately after the first node. If the conditional evaluation expression is composite, the second node might be executed immediately after the first node.

Then the cyclomatic complexity of the codes segment in figure 1 is \( V(G) = e - n + 2 = 7 - 6 + 2 = 3 \).

### 3.3.2 Advantages and disadvantages of CCM

The experience shows, there is a strong connection between the bug density and the CCM. Usually the CCM should be less than 10, and not be more than 20. When the CCM is close to 100, the software will be so complicated that, when a bug is fixed, a new bug will be introduced with the probability of more the 60%. Thus the code is nearly out of control. The CCM is proved to be effective in practice. But there are still a number of shortcomings.

Firstly, the CCM ignores the complexity from the data flow of the software. So, if we measure a module using the CCM, a sequential execution codes segment of 10,000 lines, has the same cyclomatic complexity as a single code line. So in practice, the HCM and the CCM are usually used together. The HCM is used to measure the complexity from the data flow, while the CCM measures the control flow.

Secondly, the CCM ignores the complexity added by the nesting codes. But in practice, the nesting codes are more complex. And when the code nests deeply, the difference would be tremendous.

Lastly, the CCM doesn’t distinguish the complexities of different kinds of control flow. In the CCM, the complexity of “if” is regarded as the same as that of “case”.

### 3.3.3 CCM’s variations and some analysis of them

For solving the shortcomings mentioned above, the article [17], basing upon the CCM, introduced the Pseudo-path metric model (PPMM), which makes some amendments.

On one hand, the PPMM gives different control structures different weights. The article also suggests some weights.

On the other hand, the PPMM calculates the complexity by the following formula: 

\[ V(G) = \sum_{P_i} V(P_i) \] 

The \( V(G) \) means the total complexity of the whole codes. The \( V(P_i) \) means the complexity of the \( P_i \) part of the codes, which is in the outermost space. All \( n \) parts compose the whole codes.

For the branch structure, such as “if(p){s1}else{s2}”, the formula is given as: \( V(P_i) = W_i \cdot \max(V(s1),V(s2)) \). The \( W_i \) is the weight of the branch structure. The \( V(s1) \) means the complexity of the codes segment \( s1 \) inside the branch structure. So does the \( V(s2) \).

For the loop structure, such as “while” and “for”, the formula is given as: \( V(P_i) = W_i \cdot V(s1) \). The \( W_i \) is the weight of the loop structure. The \( V(s1) \) means the complexity of the codes segment \( s1 \) inside the loop structure.

The PPMM, to some extent, makes up for the second and third shortcomings of the CCM mentioned above. But it, maybe, brings in more problems than it solves.

First of all, the weight values are questionable. The original recommended weight of the multi-branch structure is 2.5. But in practice, the “case”, a typical multi-branch structure, is considered as a kind of good structure, which doesn’t increase the complexity very much.

Secondly, the PPMM doesn’t give a reasonable and consultable range of the complexity. While, the CCM suggests, the cyclomatic complexity should be less than 15.

Finally and the most fatally, the PPMM loses the solid mathematical basis that the CCM based on.

### 3.4 The comparison of the classic metrics and their variations

At the end of this section, we give a comparison of the classic metrics and their variations, shown in the table 6.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Usability</th>
<th>Effect</th>
<th>Popularity</th>
<th>Theory Basis</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td>Easy</td>
<td>Good</td>
<td>Wide</td>
<td>No</td>
</tr>
<tr>
<td>HCM</td>
<td>Medium</td>
<td>Good</td>
<td>Wide</td>
<td>No</td>
</tr>
<tr>
<td>WHCM</td>
<td>Medium</td>
<td>Bad</td>
<td>Narrow</td>
<td>No</td>
</tr>
<tr>
<td>DCM</td>
<td>Hard</td>
<td>Bad</td>
<td>Narrow</td>
<td>No</td>
</tr>
<tr>
<td>CCM</td>
<td>Medium</td>
<td>Good</td>
<td>Wide</td>
<td>Yes</td>
</tr>
<tr>
<td>PPMM</td>
<td>Medium</td>
<td>Bad</td>
<td>Narrow</td>
<td>No</td>
</tr>
</tbody>
</table>

Almost all the metrics, except the CCM, don’t have a solid theory foundation. The current metrics of the software complexity are, somehow, based upon the experience.

### IV. THE CORRELATIONS AND CO-USE OF THE METRICS

With the metrics of the software complexity spreading, the researchers begin to study the correlations of the metrics. As early as in 1997, the article [18] realized that, there is a strong connection between the LOC and CCM.
The article [19] analyzed about 71,917 programs on 59 fields written by c/c++. The result showed there are very strong connections between LOC and HCM, LOC and CCM. The study got some approximate expressions:

Halstead Volume = 45 * LOC - 428
CC = 0.22 * LOC + 1.9

At the same time, the article [19] pointed out that, all these three metrics can’t be used to forecast the quantity of the potential bugs.

There are strong connections among the metrics. But that doesn’t mean that we can replace one metric with someone else. The article [20], using a data miner tool called Multimethod, did some experiments on three data sets in the Metrics Data Project (MDP) of the NASA. The result shows, the effect of the software defect prediction model integrating kinds of metrics, is much better than that using only one.

The article [21] showed some similar views. The article studied on 8 data sets in the MDP of the NASA. The result indicates that, the effect of the prediction model using 2 or 3 static metrics, is the same as that using 38 static metrics, but is clearly much better than that using only one metric.

At the same time, the article [21] pointed out, the best subsets of the software metrics, being used to make up the software defect prediction model, vary from one special application field to another. A recommended method in practice is that, the best subset should be selected, through machine learning, from all available metrics.

V. THE TREND OF THE METRIC OF SOFTWARE COMPLEXITY

An important trend about the metric of the software complexity is the merging several different methodologies [22]. We no more use only one metric to measure the software complexity, but use many ones together. A typical application is the co-use of the HCM and the CCM.

Another critical trend is the combination of the metrics of the software complexity and the technique of the date mining [20][21]. The method, which calculates some independent metrics of the software complexity and then analyzes and integrates them by the data mining technique, has been increasingly concerned, studied and debated.

VI. CONCLUSION

The software complexity metric is becoming an extremely important part of the software engineering. Nowadays, when more and more attentions are focused on the quality of the software, it’s reasonable to believe that, the software complexity metric will be put on its rightful place.

REFERENCES