TSINGHUA SCIENCE AND TECHNOLOGY
ISSN 1007-0214 38/49 pp212-216
Volume 12, Number S1, July 2007

A New Method for Measurement and Reduction of Software Complexity[*]

SHI Yindun (施银盾)[**], XU Shiyi (徐拾义)

School of Computers, Shanghai University, Shanghai 200072, China

Abstract: This paper develops an improved structural software complexity metrics named information flow complexity which is closely related to the reliability of software. Together with the three software complexity metrics, the total software complexity is measured and some rules to reduce the complexity are presented in the paper. To illustrate and explain the process of measurement and reduction of software complexity, several examples and experiments are given. It is proposed that software complexity metrics can be measured earlier in software development and can provide substantial information of software systems whose reliability can be modeled and used in the determination of initial parameter estimation.

Key words: software complexity metrics; information flow complexity; software reliability

Introduction

As we have known, a high qualified dependable system results from its high performance in all aspects including reliability, availability, testability, maintainability, safety, and security[1]. It is also seen that the number of residual defects in software could be one of the most important factors to decrease the quality of software and to make it unreliable. Defects can be introduced into software during its development, modification as well as testing. Therefore, the number of residual defects is closely related to the experience, skill, and psychological quality of human beings, which implies the more complicated the software is, the greater number of residual defects will be introduced into software [2]. In theory, one can find all of defects in software; however, it is impossible to perfectly realize it within a reasonable period of time. In this case, another way of keeping software with high quality should be considered. It is, for example, to reduce the complexity of software as much as possible during development so that the designer can treat it smoothly and/or directly with fewer tricks. Hence some unexpected defects can naturally be avoided when developing the software. Most researchers want to measure the complexity of the system so as to be able to manage it. Experimental results show that the costs of development and testing of software are closely related to the complexity of software [3,4]. It is believable that reducing costs and increasing dependability are the unified goals which can be achieved when the complexity of software is properly reduced[5].

Previous research on software complexity metrics has been concentrated in the following areas. Explored metrics was studied based on the lexical content of a program, including lines of code, which calculated the number of lines of code; Halstead's original work[6], which counted operators and operands; the Cyclomatic complexity measure by McCabe [7], which counted the number of branch points in a program, and other extensive study by Thayer et al.,[8] which counted the occurrence of a wide variety of statement types. These simple lexical measures have proven to be surprisingly robust in predicting various aspects of

Received: 2007-02-01
∗ Supported by the National Natural Science Foundation of China (No. 60473033)
∗∗ To whom correspondence should be addressed.
E-mail: siliver007@163.com ; Tel: 021-56332707

software quality.

Information flow complexity is an appropriate and practical basis for measuring large-scale system[9] and perhaps the most widely known metric of structural complexity, which is improved from the original work developed by Henry[10]. Observing the patterns of communication among the system components, we are in a position to define measurements for complexity, module coupling, cohesion and dependence. The information flow metric reflects the system's structure, which is different from other metrics mentioned above, with lines of code (LOC) and Halstead's volume (HV) reflecting the system's textual size mostly and Cyclomatic number (CN) reflecting the system's branches mostly. Different methods reflect different complexity factors of system.

As a matter of fact, there are a number of demerits to be desired in Henry's information flow. This metric is validated in the UNIX, but whether it can be applicable in other environment is doubtful. If either fan_in or fan_out (their definition is shown in Section 2.2) is zero, the module's information flow complexity will be zero because of the multiplication between them[11]. This condition is unpractical in the view of testing theory. The length of module in formula[10] is an independent metric in calculating the software complexity named as lines of code[12] and makes the information flow complexity correlated much to it. As a result it supplies little information for calculating total software complexity. Several different definitions of flows are used [11] and the definition of global information flows is also somewhat ambiguous[12].

Some strategies are proposed in the paper to overcome the above problems and some new definitions and methods are introduced to measure the information flow complexity in Section 2. We use lines of code, Cyclomatic number and Halstead's volume metrics together with the new information flow complexity to measure the software. The measurement and controllability of the complexity of the system are shown in Section 3.

1 Information Flow

1.1 Function call chart

We consider a function as a module and a system as a whole. The information flow complexity[13] considers a data structure as a module without thinking over the relation of different modules. If one wants to calculate the information flow complexity of a program, the only thing one does is to add the modules without considering the connection between modules. In fact, the connection between modules should not be neglected, because in the specification design stage, the data flow is extremely important and it also reflects the complexity of the program.

For example, we use the program named LOKI97 as an example whose function is encrypting and decrypting. In the program, the data structures of Plaintext, Cryptograph and Key are considered, and their relation is the foundation and main theory of system. The information flow complexity should not just add the information flow complexity of Plaintext**,** Cryptograph and Key because of a number of connections existing among them.

Reading the source code is a tough task to adequately understand the procedures of program. The data flow becomes more complicated due to the function calls. So it is reasonable to figure out the function call chart to illustrate the relation of data flow.

Function call chart can also be used to calculate the Cyclomatic Number. The branch of a program will be counted by adding every function as the information flow complexity methods described below.

Different from traditional function call chart, the purpose of function call we used is to calculate the information flow complexity. So we count the times of a function call another. The example of drawing the function call chart is given in the right of Fig. 1. The digits 2 and 1 in the right part mean FunctionA calling FunctionB twice and calling FunctionC once.

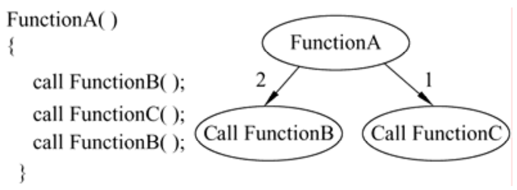

Fig. 1 Method of drawing function call chart

1.2 Measuring the information flow complexity

The term fan_in and fan_out, which were defined by Henry, are the basis of measuring information flow complexity. But the definition of them is difficult to understand and to measure. It is necessary to redefine them. We have the definition of them formally below.

Definition 1 The fan_in of procedure *A* is the number of parameter passed from outside and global variables read by procedure *A*.

Definition 2 The fan_out of procedure *A* is the number of return value and global variables written by procedure *A*.

For a procedure, the fan_in information means that there is information flowing into it and the fan_out means that there is information flowing from it to the environment, where the procedure interacts. Before starting to measure the information complexity, it is necessary to list the number of fan_in and fan_out of every procedure.

Combining the function call chart and the definition of fan_in and fan_out, the information flow complexity of system can be calculated by the following three steps.

Step 1 Calculate the information flow of every procedure without considering their position in the function call chart according to the formula,

$$(\text{fan\_in}*\text{fan\_out})^2\ [9]$$

Step 2 From down to up of the function call tree, the information flow complexity of a procedure is the information flow complexity of itself and the information flow complexity of its sub-procedure.

Step 3 The complexity of the procedure in the top of function call chart is the information flow complexity of system.

In the formula of Step 1, the term fan_in * fan_out represents the total possible number of combinations of an input source to an output destination. The weighting of the fan_in and fan_out component is based on the belief that the complexity is more than linear in terms of the connections which a procedure has with its environment. The power of two used in this weighting is the same as Brook's law of programmer interaction [14] and Belady's formula for system partitioning [15]. The information flow complexity of a system will be calculated by Step 2 and Step 3.

To explain the process of measuring information flow complexity clearly, we take a piece of LOKI97 program as an example, which is demonstrated as pseudo code.

```
makeKey(parameter p1, parameter p2, parameter p3)
{
    define local variable lv1,lv2;
    read global variable gb1;
    read p1,p2,p3;
    call procedure add(p1,p2) ;
    write global variable gb2;
    …
    return lv1;
}


add(parameter q1, parameter q2)
{
    define local_variable lva;
    read global_variable lv1;
    read q1,q2;
    …
    return lva;
}
```

As described in Definition 1 and Definition 2, the fan_in and fan_out of procedure makeKey and Add are listed in Table 1.

Table 1 The fan_in and fan_out of procedure makeKey and Add

	makeKey()	Add()
fan_in	p1,p2,p3,gb1	q1,q2,lv1
fan_out	gb2,lv1	lva

Because procedure Add does not call any user defined procedure, the IF of procedure Add is that $\text{IF(Add)} = (3\times 1)^2 = 9$. Procedure makeKey calls procedure Add, so the IF complexity of procedure makeKey is that $\text{IF(makeKey)} = (4\times 2)^2 + \text{IF(Add)} = 64+9 = 73$.

2 Measurement and Controllability of Software Complexity

There are many complications affecting software reliability module, such as software complexity, programmer skill, testing effectiveness, running environment and so on. Most researchers proposed[11] that all above complications behaved as the residual errors without considering the software complexity or test effectiveness, which is not scientific. Because there is a truism that if the software is less complicated or better test effective, there will be fewer errors in the system for fewer errors being injected and more errors being eliminated.

It is obvious that the more complicated a system is, the more errors being injected into it. If we could quantify the complexity of a system, it will be a good reference to evaluate software reliability. Because of the different characteristics of previous software complexity such as LOC, CN, HV, and the IF, we gather them together to quantify the software complexity.

For a system with four complexity methods, there would be four complexity terms and the complexity of the system can be described by a complexity vector in n-dimensional space as

$$\boldsymbol{C} = (\text{LOC, CN, HV, IF})$$

Different systems can be compared by their complexity vectors, provided that corresponding complexity metric is equal in the two systems. It is likely that each metric in the vector varies between any two systems. And it is possible to determine the differences between two systems by subtracting their complexity vectors. The result can be used to analyze their complexity, but we can compare one complexity metric of two systems according to the result vector. It cannot be used to compare the whole complexity of two systems. For example, the complexity vector of system A, $\boldsymbol{C}(A)$=(450, 65, 9000, 700) and of system B, $\boldsymbol{C}(B)$=(500, 50, 9500, 600), the subtracted vector $\boldsymbol{C}(A-B)$=(−50, 15, −500, 100) is calculated, which means the LOC and HV of system B is greater than system A and the CN and IF of system A is greater than system B. but we can not say that system A is more or less complicated than system B.

Instead, the Euclidean norm of the complexity vector is used to indicate the complexity of a system and for comparison of different systems. After calculating every complexity vector, we discover that the digit is greatly varied between different metrics. So we take every metric by logarithm.

The expression of the complexity vector is given as

$$\|\boldsymbol{C}\| = \sqrt{\ln^2\text{LOC}+\ln^2\text{CN}+\ln^2\text{HV}+\ln^2\text{IF}}$$

For example, consider the LOKI97 system and other two systems called Calculus and Library. Their complexity data is listed in Table 2.

Table 2 shows that, the level of the whole complexity according to complexity vector from high to low is LOKI97, calculus and library. By analyzing their source code, we find the reason is that in most procedure of system LOKI97, the branches and fan_in and fan_out are much greater than any other system and according to the function call chart, the relation of procedure of LOKI97 is more complicated than the other two.

Table 2 Complexity metrics of three systems

Metrics	LOKI97	Calculus	Library
LOC	1170	1138	3754
CN	119	98	179
HV	35 639	41 013	22 777
IF	51 752	1097	1171
ln LOC	7.0647	7.0370	8.2305
ln CN	4.7791	4.5849	5.1873
ln HV	10.4811	10.6216	10.0335
ln IF	10.8542	6.9763	7.0656
$\|\boldsymbol{C}\|$	300.4203	232.0304	245.2457

The complexity of a system, which was developed by different programmers, is varied dramatically. The lower the complexity is, the fewer errors remained in it. As a result, we may wonder that if there will be any regulation to follow to decrease the complexity in the phase of design specification and programming. To achieve this purpose, we can reduce the information flow complexity by two regulations.

First, in a procedure, either fan_in or fan_out should not excess 5, or else the programmer should divide it into two or more procedures. It is especially useful when calculating the information flow complexity, because

$$\begin{aligned}&(\text{fan\_in}(C)^*\text{fan\_out}(C))^2\\&=(\text{fan\_in}(A+B)^*\text{fan\_out}(A+B))^2\\&=((\text{fan\_in}(A)+\text{fan\_in}(B))^*(\text{fan\_out}(A)+\text{fan\_out}(B)))^2\\&>(\text{fan\_in}(A)*\text{fan\_out}(A))^2+(\text{fan\_in}(B)*\text{fan\_out}(B))^2\end{aligned}$$

In the formula, $A+B$ means integrating A and B in one procedure named C. For example, if the fan_in ($A+B$) is 8, fan_out ($A+B$) is 7, then the information flow complexity of procedure C IF(C) is 3136. After dividing C into A and B randomly, suppose that fan_in(A) and fan_in(B) is 3 and 5, fan_out(A) and fan_out(B) is 4 and 3, then the IF(A)+IF(B) is 369, which is much smaller than IF(C). This regulation should be used in the LOKI97 program to decrease the complexity.

Secondly, the number of branches in a procedure should not excess 10. If the number of branches in a procedure is greater than 10, the procedure should be divided into two or more sub-procedures[7]. Woodfield[16] has argued that a 25% rise in the problem complexity can lead to a 100% rise in program complexity. Usually, a procedure implements one function. If the function is very complicated, the procedure complexity will increase so quickly that more errors will be incurred. It is wise to decompose the problem into two or more sub-problems.

3 Conclusions

The development of software complexity measurement has experienced for a long time, but the criteria of measuring the software complexity have not been ubiquitously set yet. By incorporating the new concept of information flow into complexity measurements, which reflects much information of structural complexity, we may certainly make the evaluation of software complexity more accurate than ever before. Dealing with the parameters such as LOC, Cyclomatic number, Halstead's volume and information flow complexity as different components of total software complexity, we can also improve the ability to estimate software reliability. By the rules proposed in the paper, we may directly reduce every metric of software complexity. The rules can help us minimize the defects remaining in software and improve the software reliability.

References

[1] Xu Shiyi. Design and Analysis of the Dependable System. Beijing: Tsinghua University Press, 2006:53-79. (in Chinese)

[2] Khosthgftaar T M, Munson J C. Predicting software development errors using software complexity metrics. *IEEE Communications*, 1990, **8**: 253-261.

[3] Potier D, Albin J L, Ferreol R, Bilodeau A. Experiments with computer software complexity and reliability. In: Proc. 6th Int. Conf. Software Eng. IEEE Press, 1982: 94-103.

[4] Rault J C. An approach towards reliable software. In: Proc. 4th Int. Conf. Software Eng. IEEE Press, 1979: 220-230.

[5] Yin M L, Peterson J, Arellano R R. Software complexity factor in software reliability assessment. In: Proc Annual Reliability and Maintainability Symposium. IEEE Press, 2004: 190-194.

[6] Halstead M H. Elements of Software Science. New York: Elsevier, 1977:156-165.

[7] McCabe T J. A complexity measure. *IEEE Trans. Software Eng.*, 1976, **2**(4):308-320.

[8] Thayer T A, Liplow M, Nelson E C. Software Reliability. New York: North-Holland, 1978:67-79.

[9] Fenton N, Pfleeger S. Software Metrics: A Rigorous and Practical Approach. London: Thomson Computer, 2002:305-350.

[10] Henry S, Kafura D. Software structure metrics based on information flow. *IEEE Trans. Software Eng.*, 1981, **7**(5): 510-518

[11] Shepperd M, Ince D. Metrics, outlier analysis and the software design process. *Information and Software Technology*, 1989, **31**(2): 91-98

[12] Kitchenham B A. An evaluation of software structure metrics. *IEEE Trans.*, 1988: 369-376.

[13] Henry S, Kafura D. Information flow metrics for the evaluation of operating systems' structure. Ames: Iowa State Univ., 1979: 68-80.

[14] Brooks F P, Jr. The Mythical Man-Month: Essays on Software Engineering. MA: Addison-Wesley, 1975: 106-113.

[15] Belady L A, Evangelisti C J. System partitioning and its measuring. *IBM Res. Rep. RC7560,* 1979:120-131.

[16] Woodfield S N. An experiment on unit increase in program complexity. *IEEE Trans. Software Eng.*, 1979, **5**(2): 76-79.