TSINGHUA SCIENCE AND TECHNOLOGY
ISSN 1007-0214 48/49 pp266-269
Volume 12, Number S1, July 2007

Exploration of Complexity in Software Reliability[*]

CHU Yanming (褚彦明)[**], XU Shiyi (徐拾义)

School of Computer Engineering and Science, Shanghai University, Shanghai 200072, China

Abstract: Traditionally, timing and the failure rate are the only two factors considered in software reliability formula, which is actually incomplete. Reliability should be redefined as a function of software complexity, test effectiveness, and operating environment. This paper focuses on software complexity with its relation to the software reliability. Today, many software complexity measurements have been proposed, but most of them treat the reliability model incompletely. This paper proposes a new method which considers a relatively complete view of software reliability including its complexity and test effectiveness of the software being tested.

Key words: software reliability; software complexity; test effectiveness; failure rate

Introduction

With the rapid development of software engineering, the size of the software system is getting larger, and the structure more complicated. Companied with the fast growth of information industry and the internet, great changes have taken place around us. More and more people have realized the importance of security and reliability of software.

Software reliability theory is one of industry's seminal approaches for predicting the likelihood of software field failures. The common definition of software reliability is the probability that a software system will operate without failure for a specified time for specified operating conditions. Traditionally, software reliability models are expressed as

$$R(t) = \exp\left[\int_0^t -\lambda(\tau)\mathrm{d}\tau\right]$$

where τ is just the variable of integration and $\lambda(\tau)$ is the failure rate. $R(t)$ is simply the probability that a failure will occur at time t given that no failure occurred before time t [1].

Received: 2007-02-01

* Supported by the National Natural Science Foundation of China (Nos. 60473033 and 60173029)

** To whom correspondence should be addressed.
E-mail: cc_study@163.com; Tel: 86-13564442566

In this formula, only two factors are considered: timing and failure rate. It is said that its form is originally borrowed from the more mature field of hardware reliability. But hardware, as we know, wears over time, with its couplings loosening and its parts overheating. Hardware is becoming less reliable little by little and will fail from wear at last. It is obviously affected by time and environments. However, software can be perfect and remain perfect, no matter how much time you test it. Software fails because of embedded defects (man-made faults) that did not surface during development and testing. Even though there are defects in the software, how do we know the time we need to find a defect? In fact, it depends on many factors, for example, testers' experience, hardware environment, and the test effectiveness.

The traditional definition of software reliability treats every application and test case uniformly, which assumes that the testers have fully understood the application and its complexity, and that testers have applied a wide variety of tests in a wide variety of operating conditions and have omitted nothing important from the test plan.

The notions of time and the operational profile incorporated into software reliability are incomplete [2]. Reliability should be redefined as a function of application complexity, test effectiveness, and operating environment. This paper focuses on software complexity with its relation to the software reliability. So far, many software complexity measurements have been proposed, but only few of them treat the reliability model completely. This paper proposes a new method which considers a relatively complete view of software reliability including its complexity and test effectiveness in software reliability.

1 Software Complexity Measurement

In our daily life, people always believe that simple things are reliable and easy to be repaired. So does it in the design of hardware and software reliabilities. Software measurement should be applied to guiding the process of testing, because intricate and involved software may be especially difficult to code, debug, test and maintain. There are four classic approaches being used for the time being, including LOC (lines of code) [3], Halstead [4], McCabe [5] and IF(information flow) [6], which stand for software's length, numbers of operators and operands, loops and the amount of information flow, respectively.

1.1 Lines of code metric

LOC counting the size of software is usually applied to executable sentence. The major concern in the original method is the use of software size as the only complexity factor, due to the limitation of detailed information available during the early stage of a program [7]. It is likelihood that the larger the size of a program is, the more the defects there could be. The LOC metrics does represent some aspects of software complexity, but it is a rough method which does not provide the necessary fidelity for a software reliability assessment. LOC should be a referenced factor of software complexity.

1.2 Halstead metrics

This method measures program's complexity by counting the number of operators and operands. Let

n_1= number of distinct operators

n_2= number of distinct operands

N_1= total number of operator occurrences

N_2= total number of operand occurrences

Based on these components, Halstead established a formula system which is comprised of vocabulary, length, volume and so on. They are defined as

Vocabulary: $n = n_1 + n_2$

Length: $N = N_1 + N_2 = n_1 \log_2 n_1 + n_2 \log_2 n_2$

Volume: $V = N \log_2 n = N \log_2 (n_1 + n_2)$

This method is arguably correct since it dose not consider the loops and information flow that intensify the complexity[8]. For example, two programs with same lines and same Halstead value will be considered to have the same complexity. However, one has straight sequential codes, while the other has very nested loops and tricky information communication. Therefore, Halstead metrics is also a referenced factor of software complexity.

1.3 McCabe metrics

This method is proposed by McCabe in his classic paper "A complexity measure" [4] that brought forward the idea of cyclomatic complexity for the first time, and it basically measures decision points or loops of the program. This method can show the intelligibility, testability, and maintainability. Cyclomatic complexity utilizes a graph, which is derived from code. The formula can be defined as

$$M=V(G)=e-n+2p$$

where e is the number of edges, n is the number of nodes, and p is the number of connected components. In fact, we usually adopt a simple method to obtain a program's McCabe value, i.e., for a single entry and single exit module, we only need to calculate the number of decision points, and add 1 to it, then we get the McCabe value.

McCabe's cyclomatic complexity metrics measures software complexity in program's structure, but it neglects the fact that sheer length of a program is a factor of complexity.

1.4 Information flow

When it comes to software complexity, we should consider another factor: the amount of information flow between modules. The most commonly used method is fan-in and fan-out metrics. Henry and Kafura [5] defined the structure complexity as

$$C = (\text{fan-in} \cdot \text{fan-out})^2$$

where C stands for the amount of information flow in a module, fan-in is a module's inputs, and fan-out is the amount of return values and variables changed.

The data reconstruction method based on the M-estimator is then discussed by the following algorithm.

2 Relative Accurate Measurement of Software Complexity

The measurements discussed above are all relatively incomplete. Therefore, we used five programs, and calculated their complexity by the four different methods to illustrate their incompleteness. The parameters of the programs are listed in Table 1.

For convenience, all the data in Table 1 are normalized and transferred as shown in Table 2.

Then, we may display the chart of the normalized curve in Fig. 1.

Table 1 Complexity calculated by four different methods

Program	A	B	C	D	E
LOC	1170	1053	1138	1079	3754
McCabe	119	73	98	37	179
Halstead	35 639	7774	41 013	9737	22 777
IF	51 752	93	1071	672	1171

Table 2 Normalized chart for comparison

Program	A	B	C	D	E
LOC	7.064 759 028	6.959 398 512	7.037 027 615	6.983 789 965	7.444 248 649
McCabe	4.779 123 493	4.290 459 441	4.584 967 479	3.610 917 913	5.187 385 806
Halstead	10.481 195 82	8.958 540 11	10.621 644 37	9.183 688 341	10.033 520 07
IF	10.854 218 36	4.532 599 493	6.976 348 07	6.510 258 341	7.065 613 364

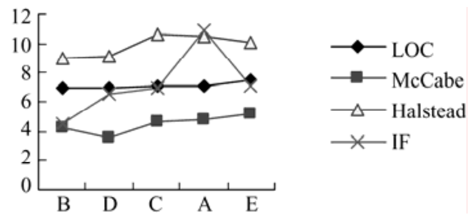

Fig. 1 Normalized distribution of complexity's factors

As is clearly shown by this graph, with the increase of LOC, other factors, including McCabe's curve, Halstead's curve, IF's curve, increase in the gross. It indicates that the program is getting complicated with the increase of length. LOC is an elementary factor affecting software complexity. Therefore, for comparison, we modified Table 1, and let LOC of every program (every column) divided by its other three factors, then getting the value/lines of other three factors in a program. The results are given in Table 3.

Table 3 The value/ lines of different factors

Program	A	B	C	D	E
McCabe/line	0.101 709 402	0.069 325 736	0.086 115 993	0.034 291 01	0.104 678 363
Halstead/line	30.460 683 76	7.382 716 049	36.039 543 06	9.024 096 386	13.320 063 4
IF/line	44.232 478 63	0.088 319 088	0.941 124 78	0.622 798 888	0.684 795 322

Table 3 shows the value/lines of the three different factors of complexity in a program. The three factors are all important to the software complexity. The larger the number is, the more complicated the program is. But they stand for different meanings.

Firstly, we must notice that the software complexity needs to be connected to the software reliability, otherwise the software complexity has no meaning. Therefore, three factors mentioned above should be all important to the software reliability. Secondly, all defects in a program are man-made defects. Hence, the three factors must be considered affecting those who program and those who test the code. Factors affecting human more are those affecting the program's reliability more, furthermore affecting the program's complexity more.

In Table 3, program A's McCabe/line is almost equal to program E's, but A's Halstead/line is two times more than E's. More operators and operands may disturb the programmer more, bringing more defects.

Similarly, more decision points and information flow may confuse the software-testers more, leaving more defects remaining in a program. Three factors are all important to the software complexity, we should take them into account completely. Therefore, a more accurate expression for software complexity could be as follows, which incorporates all factors that we mentioned above:

$$C_S = \lambda_M V_{MP} + \lambda_H V_{HP} + \lambda_{IF} V_{IFP} \quad (1)$$

where C_S is the software complexity, and V_{MP}, V_{HP}, V_{IFP} are the value/lines of the three different factors (McCabe, Halstead, and IF). λ_M , λ_H ,and λ_{IF} are their weight determined by users, $\lambda_M + \lambda_H + \lambda_{IF} = 1$, and λ_M (λ_H or λ_{IF}) $\leqslant 1$. For the five programs for this example, let $\lambda_M = \lambda_H = \lambda_{IF} = 1/3$. The normalized graphs are given in Fig. 2.

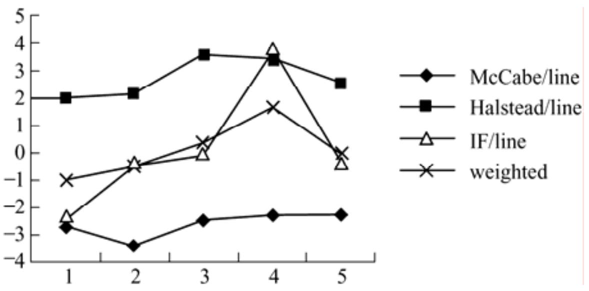

Fig. 2 Normalized weighted graph of different factors of software complexity

3 Failure Rate and Complexity

The traditional software reliability models are expressed as $R(t) = \exp\left[\int_0^t -\lambda(\tau)\mathrm{d}\tau\right]$, in which timing and failure rate are the only two factors considered. This approach is incomplete and the failure rate should be connected to application complexity and test effectiveness [1].

The common method measuring test effectiveness is fault injection, then putting the test case into the program and finding the number of defects that is injected into the program. If the test effectiveness is high, the failure rate would be the main factors affecting the software reliability. The additional factors affecting software reliability should mostly be the complexity of software [9], because the program is so intricate that other defects are hard to be detected. Therefore, we believe the proposed failure rate should be expressed as

$$\lambda_R = \lambda_e \lambda_{fr} + (1 - \lambda_e) f(C_S) \quad (2)$$

where λ_e is the test effectiveness; λ_{fr} is the failure rate of the program; C_S stands for software complexity, and $f(C_S)$ is a function with respect to C_S. In this case, we believe that the new failure rate, λ_R , could be more comprehensive than we used to use in evaluating the reliability of software. Equation (2) indicates that software reliability is not only related to the time, but also close to the application complexity, the test effectiveness, and other factors.

4 Conclusions

Today, software is becoming more complicated with more fancy functions. Suitable methods should be proposed to express their real reliability. This paper brought forward a relatively more comprehensive method measuring software complexity which heavily influences the evaluation of reliability of software. Furthermore, the paper also proposed a formula of software reliability, which is connected to the test effectiveness and application complexity. Meanwhile, the work is desirable to be improved in the near future.

References

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

[2] Whittaker J A, Voas J. Toward a more reliable theory of software reliability. *Computer*, 2000, **33** (12): 36-42.

[3] Kan S H. Metrics and Models in Software Quality Engineering (2nd Edition). Boston: Addison-Wesley Professional, 2002.

[4] Halstead M. Elements of Software Science. New York: Elsevier North-Holland Inc, 1997.

[5] McCabe T J. A complexity measure. *IEEE Transactions on Software Engineering*, 1976, **2** (4): 308-320.

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

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

[8] Anneberg L, Singh H. Circuit theoretic approaches to determining software complexity. In: Circuits and Systems, Proceedings of the 36th Midwest Symposium on. 1993: 895-898.

[9] Lew K S, Dillon T S, Forward K E. Software complexity and its impact on software reliability. *IEEE Transactions on Software Engineering*, 1988, **14**(11): 1645-1655.