

A resource model of complexity (RMC)

A. Törn, T. Andersson, K. Enholm

*Abo Akademi University, Computer Science Department,
SF-20520 ABO, Finland*

ABSTRACT

In this paper we will review proposed software complexity measures and explore what concepts might lay behind them. We will also look at proposed axioms of complexity measures and try to deduce what complexity concepts would comply with these. The result of this study will then be discussed relative to the length-independent measure RMC defined by $w(P) = l(P)c(P)$, where P is any piece of software, $w(P)$ the effort in developing it, $l(P)$ its length, and $c(P)$ its complexity. Application of RMC to both structured programming constructs and to collections of programs is discussed.

INTRODUCTION

Several software complexity measures correlate strongly with software length measures. As length correlates with effort or cost the complexity measures could thus be used to measure cost. However, then the length measure could be used instead. Measures should basically be independent and therefore we think that a complexity measure should be independent of software length.

Compare with the situation when buying some amount of some item. The entities involved are amount (measured in number of items n), unit price (number of money units/item p_u) and total price (p_t). In this case the independent measures are n and p_u and from these the indirect measure p_t can be calculated. In analogy to this, for RMC complexity is comparable to p_u , length to n and effort to p_t . We thus propose that complexity is measured per unit length of software which means that our complexity measure can be regarded as average complexity over length for software

174 Software Quality Management

of length differing from one.

In the section *Complexity Axioms* we will discuss what sort of measure is implied by some proposed axiomsystems describing desirable properties for software complexity.

In the section *Complexity Measures* we will analyze different proposed complexity mesures. We are interested to find out if the measures correlate with the length measure.

In the section *RMC* the new measure is described in detail and its most important properties are stated. The measure is defined for collections of programs (systems, libraries) and for the structured constructs used as building blocks in programs.

COMPLEXITY AXIOMS

Most of the discussion in this section is based on the material presented in Zuse [11].

The following axiom is included in Bache's axioms of complexity [2]:

$$\mu(F; G) > \max(\mu(F), \mu(G)).$$

This says that the complexity of a concatenated program is always larger than the complexity of either part regardless of the complexities or lengths of the parts.

The desirable properties by Weyuker [10] include:

Property 4: $(\forall P)(\forall Q) (\mu(P) \leq \mu(P; Q))$ and $(\mu(Q) \leq \mu(P; Q))$. This is the same as Bache's axiom where $>$ is replaced by $\geq$.

Among the required properties proposed by Jayaprakash *et al* [8] there is the *Additive Property* is stated as:

"Let s_1 and s_2 be two one-entry, one-exit arbitrary pieces of program code with control flow complexities m_1 and m_2 . Now if a large program is constituted by placing s_1 and s_2 in sequence, the resulting program should ideally have a control flow complexity of $m_1 + m_2$, because sequence structuring does not involve any additional complexity. Observe that when the control flowgraph corresponding to the program segments s_1 and s_2 are combined the exit-node of s_1 will be merged with the entry-node of s_2 . Hence no new edges or nodes are effected."

All three axioms imply that complexity increases with length of software. This is consistent with the view of complexity interpreted as effort to develop the software.

In [7] Fenton discusses implications of measurement theory on complexity relations on flowgraphs based on the requirement that such a relation in order to be at least ordinal must be negative transitive. Then he presents an example which illustrates the impossibility to define a consistent measure. The example involves comparing flowgraphs representing a single choice (y), two sequential choices (x) and an iteration (z). He states xCy (x is more complex than y) but that neither xCz nor zCy is valid contradicting transitivity. However it seems that complexity is interpreted as length in the first case whereas for the negative conclusions that complexity means something else. Using the RMC approach above we conclude that the first two (y, x) have the same complexity (the length doubles) and therefore no contradiction exists if RMC is used.

COMPLEXITY MEASURES

One of the first software quality measures proposed is McCabe's cyclomatic complexity measure. As described by McCabe, the primary purpose of the measure is to "... identify software modules that will be difficult to test or maintain" [9].

McCabe's complexity measure is defined for a flowgraph to be

$$c(F) = e - n + 2,$$

where e is the number of edges and n the number of nodes of F . Concatenating two identical flowgraphs by making the end node of the first to be the start node of the second we obtain

$$c(F; F) = 2(e - n) + 3 = 2 \times c(F) - 1,$$

which means that McCabe's measure normally is dependent on length. The only exception is when $c(F) = 1$ in which case $c(F; F) = 1$. Then F is a number of sequential nodes. Only decision nodes count and McCabe's measure is therefore more or less a count of the decision nodes. As normally a longer program will have more decision nodes McCabe's measure thus strongly correlates with length. In Grady [5] it is reported that for a system, the number of updates was proportional to the number of decision statements. This finding is of course not surprising, because it means that the number of updates is proportional to program length.

Halstead defined and discussed several complexity measures based on the source code of programs [6]. The direct measures used are:

Number of distinct operators: $n1$

Total number of operators: $N1$

176 Software Quality Management

Number of distinct operands: n_2

Total number of operands: N_2 .

Based on these the following measures are defined:

Length $N = N_1 + N_2$,

Vocabulary $n = n_1 + n_2$,

Volume $V = N \log n$,

Level V^*/V , where V^* is the minimal *Volume* implementation,

Difficulty $D = 1/L$, measured as $(n_1/2)(N_2/n_2)$, and

Effort $E = VD$.

Of these most correlates with length. An exception is D (L). In D N_2/n_2 is the average number of occurrences in the program of each operand which could be independent of length. However, according to the definition *Difficulty* rather means difficulty in finding a solution to the problem than difficulty in understanding the code.

There are reports that McCabe's measure and Halstead's measure effort have correlations of the order 0.9 [3]. The explanation is that they both correlate with length.

In Banker *et al* [3] the high correlation of cyclomatic complexity with lines of code is given as reason for proposing a transformed metric, *density of decision making*, obtained by dividing the cyclomatic complexity with the total number of statements, and Gill and Kemerer [4] use 'complexity density' defined as the ratio of cyclomatic complexity to thousand lines of code. By this they want to make the complexity measure independent of length. This is well in line with our approach.

RMC

We assume that complexity of a program is a entity of density type [1]. Using the analogy with mass (m), volume (v) and density (d) in Physics we know that for an object A

$$m(A) = v(A)d(A).$$

For a piece of software P the software mass w could analogously be defined as

$$w(P) = l(P)c(P), \quad (1)$$

where l is the length of the software and c is the complexity, giving

$$c(P) = w(P)/l(P). \quad (2)$$

Having two pieces of software P and Q with given lengths and complexities the length and complexity of the collection $[P, Q]$ could then analogously be defined as

$$l[P, Q] = l(P) + l(Q)$$

and

$$c[P, Q] = \frac{l(P)c(P) + l(Q)c(Q)}{l(P) + l(Q)}, \quad (3)$$

i.e., the complexity of a collection of programs is the length weighted average of the complexities of its parts.

Properties of RMC

We list the properties derived in [1].

The scale for c is ratio.

For RMC when concatenating items the complexity of the resulting item is always inside the interval of complexities determined by the complexities of the components. This follows easily from the interpretation of (3) as a weighted average. We thus have

$$\min[c(P), c(Q)] \leq c[P, Q] \leq \max[c(P), c(Q)]. \quad (4)$$

This is the property of RMC that makes it differ from most other complexity measures. RMC obeys commutativity, which follows directly from formula (3), i.e.

$$c[P, Q] = c[Q, P]. \quad (5)$$

This follows naturally from the interpretation of (3) as a weighted average.

RMC for the structured program constructs

Based on the definition of complexity for collections of software the average complexity for different collections (systems, libraries) could be calculated based on the length and complexities of the individual programs contained in the collection.

In this section we will extend the approach to structured programming constructs so that the complexity of a program could be computed if knowing the length and complexities of its parts.

We define the complexity of a concatenated program to be the same as for a collection, i.e.,

$$l(P; Q) = l(P) + l(Q)$$

and

$$c(P; Q) = \frac{l(P)c(P) + l(Q)c(Q)}{l(P) + l(Q)}. \quad (6)$$

178 Software Quality Management

For the choice construct: *if B then P else Q* we define

$$l(if) = l(B) + l(P) + l(Q)$$

and

$$c(if) = c_{if}c(B; P; Q),$$

where c_{if} is some constant > 1 which then determines the relative difference in complexity between sequence and an if-statement.

For the iteration construct: *while B do P* we define

$$l(while) = l(B) + l(P)$$

and

$$c(while) = c_w c(B; P),$$

where c_w is some constant > 1 which then determines the relative difference in complexity between sequence, an if-statement and a while-statement.

Of course letting c_{if} and c_w be constants is only a first approximation. There could be a range for them and the user could then make an explicit choice based on his evaluation of the complexity of the actual construct.

The effect of using constants $c_{if}, c_w > 0$ is that nesting will increase complexity. Assume that in an program A consisting of an *if* statement the *else* part Q is replaced by a call to a module Q' performing the same computation as Q . Let us name this new program A' . Let us compare the complexity of A with the complexity of the collection (A', Q') . We have $c(A) = c_{if}c(B; P; Q)$ and

$$c(A', Q') = \frac{c_{if}(l(B)c(B) + l(P)c(P) + l_{call}c_{call}) + l(Q')c(Q')}{l(B; P; Q') + l_{call}},$$

where l_{call} and c_{call} are the length and the complexity of a subroutine call. Assume that $l(Q) = l(Q')$ and $c(Q) = c(Q')$. Then if l_{call} is negligible compared to $l(B; P; Q)$ the modularization is bringing down the complexity if

$$l(Q)c(Q) > \frac{c_{if}l_{call}c_{call}}{(c_{if} - 1)}.$$

This means that a modularization pays if the effort in writing Q is large enough. This will be true for some large enough value of $l(Q)c(Q)$ even if the simplifying assumptions are not made. This result regarding modularization is according to our intuition.

CONCLUSIONS AND DISCUSSION

The proposed resource model for complexity (RMC) clearly separates the two components length and (length-independent) complexity which are parts of most existing complexity metrics. This separation leads to a cleaner concept and resolves some contradictions which must otherwise exist. We also argue that the new concept better conforms with our empirical understanding of the word complexity.

Using the RMC concept it is possible to approach the more general concept Information System Complexity. The complexity concept can be applied to subsystems and systems by using the formula for collections.

It also gives a possibility to calculate the complexity measure c as an indirect measure by using measures for l and w giving the unit for complexity as *effort/line*. The measure could be used to identify software with high complexity. High complexity could be caused either by high complexity of the problem solved or that a sophisticated solution was implemented. If neither of these explanations apply then a high complexity measure could indicate low quality software.

We have to some extent investigated the possibility of defining a direct measure compatible with RMC. The results are roughly intuitively correct. Further work is needed to assert the usefulness of this approach. This includes fixing the length and complexity for atomic constructs (assignment, expressions, etc.), and fixing relational constants such as c_{if} , c_w by making validating experiments.

REFERENCES

1. Andersson, T., Enholm K. and Törn A. 'A Length-Independent Measure of Software Complexity', In: (Ed. Ross M, Brebbia C.A., Staples G, Stapleton J.) pp. 149-157, *Software Quality Management II, Vol. 1, Managing Quality Systems*, Computational Mechanics Publications, Southampton Boston, 1994
2. Bache, R. 'Metricating A-KINDRA BT Project 610287', Document W021, Suuth Bank , 1987.
3. Banker, D.D., Datar S.M. and Zweig D. 'Software complexity and maintainability', In: (Ed. DeGross J.I., Henderson J.C., Konsynski B.R.), pp. 247-255, *Proceedings of the Tenth International Conference on Information Systems*, Dec. 4-6, Boston, 1989
4. Gill, G.K. and Kemerer C.F. 'Cyclomatic Complexity Density and Software Maintenance', *IEEE Trans. Software Engineering*, Vol. SE-17, 1284-1288, 1991



180 Software Quality Management

5. Grady, R.B., 'Successfully applying software metrics', *Computer*, Vol. 27, No. 9, 1994.
6. Halstead, M.H. *Elements of software science*, Elsevier North-Holland, New York, 1977
7. Fenton, N.E. *Software Metrics – A Rigorous Approach*, Chapman & Hall, London, 1992
8. Jayaprakash, S., Lakshamanan, K.B., Sinha, P.K. 'MEBOW: A Comprehensive Measure of Control Flow Complexity', COMPSAC 87, 238-244, 1987.
9. McCabe, T. 'A complexity measure', *IEEE Transactions of Software Engineering*, Vol. SE-1, 312-327, 1976
10. Weyuker, E.J. 'Evaluating software complexity measures', *IEEE Transaction of Software Engineering*, Vol. SE-14, 1357-1365, 1988
11. Zuse, H. *Software Complexity*, de Gruiter, Berlin, 1991