

Length-independent measure of software complexity

T. Andersson, K. Enholm & A. Törn

*Åbo Akademi University, Computer Science
Department, SF-20520, ÅBO, Finland*

ABSTRACT

A software complexity concept based on measurement theory is introduced. It is argued that a complexity measure conforming with our empirical understanding of complexity should be independent of software length. In analogy with the formula for mass, volume and density the measure is 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. The properties of the measure are discussed and it is found that some of the conflicts in desirable properties reported for other complexity measures are resolved.

INTRODUCTION

Software complexity is an attribute with many meanings that is used to characterize different entities. In many cases complexity is defined in such a way that it correlates strongly with the length of a program. We argue that complexity should be an attribute that measures something independent of length.

We first present approaches to complexity as found in the literature. Based on a discussion of this we propose a measure theoretic approach to complexity. Measure theory will therefore be briefly introduced and an empirical understanding of the concept complexity will be presented. Then a resource model of complexity is presented and discussed.

COMPLEXITY IN THE COMPUTER LITERATURE

The results of a literature search on software complexity is reported below.

350 Software Quality Management

The literature survey is by no means complete but is meant to give a representative picture of the different meanings of the word complexity.

Complexity Concepts

For IS or software the word complexity was probably first used for what is called *computational* or *time complexity*. As an example the task of searching a sorted list of length n for a single item has complexity $O(\log n)$ meaning that any algorithm giving a solution to the task will in the worst case need the order $\log n$ pairwise comparisons to solve the task for large n . The task to sort such a list has computational complexity $O(n \log n)$ and is thus a more complex task. These complexities characterizes the class of problems to be solved and gives the growth in computation time as a function of the growth in problem size. In addition to this each method designed to solve problems belonging to some class has its own complexity which of course cannot be less than the complexity of the corresponding problem class.

Complexity has also been used to characterize software. According to McCall [6] complexity “relates to data set relationships, data structures, data flow, and the algorithm being implemented” and “measures the degree of decision-making logic within the system”.

Beizer [2] states that “using only our intuitive notion of software complexity, we expect that more complex software will cost more to build and test, and will have more latent bugs”.

Tourlakis [10] distinguishes between two classes of complexity measures, *dynamic* and *static*. Dynamic complexity measures are measuring the amount of “resources” consumed during a computation. Static complexity measures on the other hand may be the size (e.g., program length) or the structural complexity (e.g., level of nesting of do-loops) of an algorithm’s description.

Ramamoorthy *et al* [7] states: “We define software complexity as the degree of difficulty in analysis, testing, design, and implementation of software. We will not attempt to attach a single number to software complexity. Instead, we discuss the complexity of individual characteristics of software”.

Jones [5] in his discussion on measuring programming complexity identifies “two logically distinct tasks: (1) measuring complexity of the problem, i.e. the functions and data to be programmed; and (2) measuring the complexity of the solution of the problem, i.e. the software itself”.

Shepperd [8] writes: “Complexity is a metaphysical property and thus

not directly measurable. What is required is a means to link the behaviour of the product characteristics that are measurable”.

Banker *et al* [1] states “Software complexity refers to the extent to which a system is difficult to comprehend, modify and test, not to the complexity of the task which the system is meant to perform; two systems equivalent in functionality can differ greatly in their software complexity”. They notice that most complexity metrics proposed confound the complexity of a program with its length. They also propose to measure length-independent complexity metrics by measuring “density of decision-making” and “density of branching” within a program.

In Gill and Kemerer [3] the high correlation of cyclomatic complexity with lines of code is given as reason for proposing a transformed metric “complexity density” defined as the ratio of cyclomatic complexity to thousand lines of code.

Zuse [13] agreeing with Ramamoorthy and Shepperd says that “the term *software complexity* is still not well defined. Here the term complexity measure is a misnomer. ... It deals with the psychological complexity of programs. ... The overall complexity of software is a function of many factors. In literature we can find many types of measures, for example process measures, product measures, resource measures, static measures, descriptive measures, black-box measures, quality measures, code measures, design measures, inter- intra-modular measures, data flow measures, information flow measures, and specification measures. The complexity of a module or a program system is influenced by the factors of cohesion, coupling, decomposition, and intra-modular complexity. It can be said that the measurement of complexity is synonymous with determining the degree of difficulty in analyzing, maintaining, testing, designing, and modifying software”.

Fenton [4] states that “Complexity is commonly used as a term to capture the totality of all internal attributes. When people talk of the need to control complexity what they really mean is the need to measure and control a number of internal (structural) product attributes”. He also states that “There appears to be three such distinct [*orthogonal* and *fundamental*] attributes of software: length, functionality, and complexity of the underlying problem which the software is solving”.

Discussion

From the exposition above we see that there is a variety of ways in which complexity has been defined. Several authors are aware about the problem that defined complexity measures correlate strongly with the length of software. They also point out that complexity correlates with cost.

352 Software Quality Management

Further they find that complexity of software has to do with actions to be performed on software.

Much of the approaches are *ad hoc*. In order to more systematically approach the complexity concept we feel that such an approach should rely on measurement theory. We will therefore next briefly introduce measurement theory [4,13].

MEASUREMENT OF ATTRIBUTES OF ENTITIES

When we are using attributes of some entities it may be the case that for each entity we can say if the entity has the characteristic or not (e.g. is complex). More often it may be the case that the entity has the characteristic to some degree. This means that given two entities we may conclude that one of the entities has more of the attribute than the other entity (e.g. *A* is more complex than *B*). Such an empirical relation system is the base for measurement of the attribute and requires that we have a clear conception of the class of entities and the attribute in question.

A measurement means mapping the entities on some (numerical) scale with respect to the characteristic of interest. The mapping produces a numerical relation system. The mapping is useful if the relations of the empirical relation system are preserved in the numerical relation system. This is called the *representation condition*. If this is valid we say that the mapping is a measure for the attribute. The measure is normally not the only one possible but defined up to some transformation that will preserve the representation condition *admissible transformations*. The type of this transformation defines the scale of the measure. The possible scales and their admissible transformations are: the scale may only classify the entities (nominal scale, any one-to-one mapping), permit comparisons (ordinal scale, strictly increasing function), permit assigning a relative measure of the attribute of an entity not including entities with zero measure (interval scale, linear transformation $\alpha x + \beta$, $\alpha > 0$), and including entities with zero measure (ratio scale, αx , $\alpha > 0$). When the measurement is performed by counting giving an integer as measure the measure is the only possible (we talk about an absolute scale, the only transformation possible is the identity transformation).

A statement containing numerical measures is *meaningful* if its truth (or falsity) remains unchanged under scale transformations. This means that the scale chosen will dictate for instance what statistical central measures like the mean, the median or the mode that can be used for the measured attribute of the entity of interest.

In the treatment of complexity in the literature presented in the pre-

ceeding section these measurement theoretic implications have not always been adhered to. However, if we want to use measurement as the powerful tool it is in many other areas we must not ignore theory.

Further on there is one thing that we want to point out. As stated above the class of entities of interest and the attribute to measure must of course be clearly understood. This is very important for the representation condition to be meaningful. For the entity being a process complexity trivially means the complexity of some activity and the resources needed to perform that activity. However, this is true even for the entity being a product. For instance something is complex because it is difficult to *understand* or to *modify*.

A SIMPLE RESOURCE MODEL OF COMPLEXITY (RMC)

In this section we first discuss how to understand the word complexity. Based on the reached empirical understanding of complexity a new complexity concept is introduced and its properties discussed.

Empirical Understanding of Complexity

A general definition of complexity is given in Webster's New Dictionary of Synonyms [11]: "Something is *complex* which is made up of so many different interrelated or interacting parts or elements that it requires deep study or expert knowledge to deal with it".

As stated above complexity has to do with the amount of resources that is required to perform some activity. The more resources that must be spent to achieve something the more complex is the entity with respect to this task. However, the definition and our intuition tell us that the amount of resources used is not a sufficient characteristic for classifying a task as complex. Even if the construction of two miles of highway will cost approximately two times the cost to construct one mile of highway the complexity of the two tasks may be regarded approximately equal because the former may not involve a larger number of different interrelated parts. This means that complexity and length are different characteristics of the task and that the resources needed to accomplish a task is a function of the size of the task and its complexity. Of course complexity may increase as a function of the size of the task but this increase should normally be less than the increase in size.

Our conclusion that complexity should be measured independently of length is not unique, see [1,3] in the literature survey. What we want to do is to define such a measure in a measure theoretic way and to motivate the approach by discussing the properties of the measure.

354 Software Quality Management

The RMC Complexity Model

Let us thus assume that complexity of a program is a entity of density type. 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

$$c(P) = W(P)/L(P). \quad (2)$$

For concatenated software $P; Q$ we then obtain

$$c(P; Q) = W(P; Q)/L(P; Q). \quad (3)$$

But the masses are additive so that

$$W(P; Q) = W(P) + W(Q) = L(P)c(P) + L(Q)c(Q), \quad (4)$$

and the same is true for the lengths and therefore we obtain

$$c(P; Q) = \frac{L(P)c(P) + L(Q)c(Q)}{L(P) + L(Q)}, \quad (5)$$

i.e., the complexity of a concatenated program is the length weighted average of the complexities of its parts. How could then software mass be interpreted? It is an entity that better than length expresses the size of the software. The size (value) of software could be measured by the effort (cost) spent in developing the software [9]. Costs are additive so this is in accordance with the assumption above about the additivity of software masses.

Properties of RMC

Is then such a model of complexity consistent with our empirical understanding of complexity? We first address the problem of determining the scale. The scale of length and mass are ratio so the rescalings for L and W are

$$L' = \alpha L \text{ and } W' = \beta W, \text{ where } \alpha, \beta > 0.$$

This gives

$$c' = \beta W / \alpha L = \gamma c, \text{ where } \gamma = \beta / \alpha > 0.$$

This means that the scale for c is ratio. The ratio scale implies that there exist programs with zero complexity. In analogy with density in Physics this would be programs with empty lines as body. The question whether such programs also have zero length depends on how length is measured. When length and complexity are separated as in RMC the length could really be measured just as number of lines. By increasing the length and readability by commenting and generous spacing the complexity would decrease so that the mass would remain the same. This seems to be well in line of what we expect from a complexity measure.

Several authors have proposed axioms or properties that a complexity measure should satisfy or have. We will mainly comment on some of the properties proposed by Weyuker [12]. For a rather covering treatment, see Zuse [13].

Many of the axiom systems contain axioms of the type *it is not true that P is more complex than $P;Q$* , which is consistent with the view that complexity is related to length. Such an axiom is then in conflict with other axioms exhibiting another view of complexity. This is pointed out by Fenton [4] who refers to Weyuker's axioms for complexity measures. Property 5 is that concatenation does not decrease complexity. Property 6 is that when concatenating program P and Q of equal complexity with the same program R the resulting programs $P;R$ and $Q;R$ may have different complexity. Here property 5 depends on viewing complexity as a function of length and property 6 as something else. In RMC property 5 should not be true because complexity is defined independent of length.

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 (5) as a weighted average. We thus have

$$\min[c(P), c(Q)] \leq c(P; Q) \leq \max[c(P), c(Q)]. \quad (6)$$

This means that RMC does not have property 5 and this is only natural for RMC. How is it then with property 6? We have for $c(P_1) = c(P_2) = c_0$ that

$$c(P_i; R) = [L(P_i)c_0 + a]/[L(P_i) + b], \quad i = 1, 2$$

where a and b are some constants. This gives

$$c(P_1) = c(P_2) \wedge L(P_1) \neq L(P_2) \Rightarrow c(P_1; R) \neq c(P_2; R), \quad (7)$$

which means that RMC has property 6.

By further examining Weyuker's axioms property 7 says that permuting the statements of a program may change the complexity. This is

356 Software Quality Management

not true for RMC which obeys commutativity, which follows directly from formula (5), i.e.

$$c(P; Q) = c(Q; P). \quad (8)$$

This follows naturally from the interpretation of (5) as a weighted average.

Property 9 states that the sum of complexities of two programs is less or equal to the complexity of the concatenated program. This statement is meaningless for RMC because the sum of two complexity measures is not tied to any program and therefore has no interpretation in the empirical relation system. As shown above (formula (6)) 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.

The other properties of Weyuker 1, 3, 4, 8 are trivial and the merit of requiring 2, ("there are only finitely many programs of complexity c_0 ") questionable.

In [4] Fenton gives a counter example for a complexity relation on flowgraphs based on the requirement that such a relation in order to be at least ordinal must be negative transitive. The counter example involves comparing flowgraphs representing a single choice (y), two sequential choices (x) and an iteration (z). For the result xCy (x is more complex than y) it seems that complexity is interpreted as length whereas for the conclusions that neither xCz nor zCy complexity means something else. Using the approach above we conclude that the first two (y, x) have the same complexity (only the length differs) and therefore no contradiction exists.

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. RMC also gives a possibility to calculate the complexity measure c as an indirect measure by using measures for L and W (formula (3)) 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.

It would be interesting to investigate the possibility of defining a direct measure compatible with RMC. Also using the RMC concept it would

be interesting to approach the more general concept Information System Complexity. The complexity concept may be applied to subsystems and systems by using the formula for concatenated software, i.e., the complexity of a system S consisting of two programs $P1$ and $P2$, $S = \{P1, P2\}$ could be computed as $c(S) = c(P1; P2)$. We plan to address these questions in future papers.

REFERENCES

1. 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
2. Beizer, B. *Software system testing and quality assurance*, Van Nostrand Reinhold, New York, 1984
3. Gill, G.K. and Kemerer C.F. 'Cyclomatic Complexity Density and Software Maintenance', *IEEE Trans. Software Engineering*, Vol. SE-17, 1284-1288, 1991
4. Fenton, N.E. *Software Metrics - A Rigorous Approach*, Chapman & Hall, London, 1992
5. Jones, C. *Programming Productivity*, McGraw Hill, New York, 1986
6. McCall, J.A., Richards P.K. and Walters G.F. *Factors in software quality*, Volumes I-III, US Rome Air Development Center Reports NTIS AD/A-049 014, 015, 055. National Technical Information Service, US Department of Commerce, 1977
7. Ramamoorthy, C.V., Tsai W-T., Yamura T. and Bhide A. 'Metrics guided methodology', *COMPSAC 85*, pp. 111-120, 1985
8. Shepperd, M. 'An evaluation of software product metrics', *Information and Software Technology*, Vol. 30, No. 3, pp. 177-188, 1988
9. Törn, A.A. 'Models of software accumulation', *J. of Systems & Software* Vol. 12, 39-42, 1990
10. Turlakis, G.J. *Computability*, Reston, Virginia, 1984
11. *Webster's New Dictionary of Synonyms*, Meriam Company Publishers, Springfield, USA, 1978
12. Weyuker, E.J. 'Evaluating software complexity measures', *IEEE Transaction of Software Engineering*, Vol. SE-14, 1357-1365, 1988
13. Zuse, H. *Software Compeixity*, de Gruiter, Berlin, 1991