An Axiomatic Evaluation of Software Complexity Metrics for Prototype based Object Oriented Systems

Syed Ahsan

Faculty of Computing and Information Technology King Abdulaziz University, North Jeddah
Email: saehsan@kau.edu.sa

Abstract: Traditional Software metrics are both tangible such as LOC and intangible such as Amount of Effort and Software Engineering experience etc. For contextual and intangible metrics, it is generally assumed that these metrics can be generalized over a universal set of domains. In our opinion, the intuitive nature of intangible and intrinsically intuitive metrics warrant a revisit of theoretical foundations on which such metrics are based. This is especially true in case of Prototype based software modeling, which being a class-less system has entirely different theoretical foundations. Also the sizes of the project on which such metrics are applied have a bearing on complexity. In this paper we have proposed an axiomatic approach towards evaluation of Software Complexity metrics. These axioms are based on Measurement theory and provide us with theoretical underpinnings based upon which, metrics suitable for class based and class-less Object Oriented approaches can be defined. Also introduced in this paper is the notion of a viewpoint which helps us to resolve the problems introduced because of intuitive contextual nature of complexity.
[Syed Ahsan. **An Axiomatic Evaluation of Software Complexity Metrics for Prototype based Object Oriented Systems.** *Life Sci J* 2014;11(3s):196-201]. (ISSN:1097-8135). http://www.lifesciencesite.com. 29

Keywords: Measurement Theory, Class-less objects, Complexity Metrics, Empirical Relation, Software Engineering

1. Introduction

Software Metrics can be perceived as a number, a measurement scale, an identifiable attribute or an empirical model depending on what is being measured [1]. For factors such as LOC (Line of Code), Amount of Effort and Software Engineering experience etc., it may quantify into a number [2],[3]. For something contextual and intangible, it may be a scale of measurement. It may be used to identify a wanted or unwanted attribute such as cohesion, portability or coupling.. We argue that complexity is intrinsically intuitive and relative. Software attributes cannot be precisely or completely defined by a conventional absolute metrics [4]. The case becomes starker in case of Prototype based software modeling, which being a class-less system has entirely different theoretical foundations [5]. Secondly, term metric has been borrowed from measurement theory. The conventional metrics of software engineering has also been designed, developed and tested in context of measurement theory. It will be erroneous to assume that any such metric would suffice for any situation in a variety of domains. Such an approach leads us to believe that such metrics exist that are universally validated and tested. On the contrary, such metrics do not exist especially for Prototype based object modeling techniques. On the same lines, we believe that a metric that caters suitably to small OO projects cannot be directly applied to large OO System without further enrichment.

Measurement Theory is being increasingly recognized as having a major contribution towards the maturity of Software Engineering metrics [6][4]. Axiomatic approaches are also useful but need further modifications and enhancements to their current form [4]. These two approached are significant in formalization and evaluation of OO metrics. In this paper we provide theoretical underpinnings based upon which, metrics suitable for class based and class-less Object Oriented approaches can be defined. These theoretical underpinnings will be based on measurement theory which we will briefly underline next. In addition to measurement theory, we will also introduce the idea of a viewpoint. A viewpoint is an empirical relationship between the viewers and the object under observation. In simple terms, it is a way of looking at something. We feel that any measure should be dependent on at least one attribute of the object under observation. Also the attribute on which the measure is based should be precisely defined. As OO systems encompass both structure and data, a measure should be able to measure both. Lastly, a measure should be reliable and automatable. A measure is reliable if it gives consistent results irrespective of the order and magnitude.

2. Measurement Theory and Qualitative Relational System

Measurement theory which provides basis for software quality metrics subsumes the notions of Scales and Measure and also the methods and procedures of measuring [7]. We will also briefly

discuss the topics of homomorphism and atomic modifications. Concepts of Meaningfulness, Weak order and extensive structures will also critically analyze. Measurement theory states that any relationship between two objects in our problem domain can be empirically observed and formalized [8]. Using this argument we stress that within the domain of SE, if complexity is relative and contextual and its meaning depending on the observer, than any complexity, for example cyclomatic complexity, cannot be formalized. Also if we have a precise and unique definition of cyclomatic complexity, then any measure of it from different observers should yield the same result. For example, given two pipes of length a and b, we either have a $\geq$ b or b $\leq$ a. Similarly a $\circ$ b would mean two pipes joined end to end($\circ$ as a concatenation operator).

Formally

$$A = (A, R_i, \circ_j) \; for \;\; i = 1 \; to \; n, j = 1 \; to \; n$$

where

A is a non empty set of objects, R_i is an empirical relationship ($\geq$, $\leq$) $\circ_j$ is a binary operation such as concatenation. Expressing binary operation as a

$$(n + m - 1) - \; tuple \; (A, R_1, R_2, \ldots, R_n),$$

We define a mapping function $\boldsymbol{\mu}$ which provides homomorphism $: \mu A \rightarrow B$ where B is a formal relational system defined as :

$$B = (B, S_i, \bullet_j) i = 1 \; to \; n, j = 1 \; to \; n$$

where B is a non empty set of objects S_i represents relationships B can enter into ($\geq$, $\leq$) and $\bullet_j$ defines closed binary operations (addition, multiplication). The homomorphism: $\mu A \rightarrow B$ then results in

$$\forall \, a \in A \, \exists \, B \in | \, \mu \, (a) = b$$

for all objects of A ($a \in A$). Formally

$$\mu \, (ab) = (\mu \, a)(\mu \, b) \;\; \forall \, a, b \in A$$

Finally we define scale as $triplet$ (A, B, μ) if and only if :

$$R_i \, (a_{ij}) \leftrightarrow S_i \, (\mu \, (\, a_{ik} \,))$$

and

$$\mu \, (a \circ_j b) = \mu \, (a) \bullet_{ju} (b)$$

for all i, j, and for all $a_{ik} \in A$. Moreover, $B = R$ (set of real numbers) implies that $triplet$ (A, B, μ) is a real scale. In our opinion, two issues that affect software complexity metrics in relation to measurement theory are: i) There is no agreed upon measure of factors such as software size or software complexity. This makes it difficult to uniquely gauge two soft-wares as equal or one greater / smaller than the other. ii) Determination of scales which are suitable for conventional software engineering metrics. Although certain size measures such as LOC may be more generic, it is difficult that all software engineers will award the same ranking for particular software as this ranking will be intuitive and will depend upon their knowledge. It may be termed as contextual. In other words, this lack of any agreed intuitive understanding of what is complex software, it is impossible to arrive at any empirical relational system. In absence of any empirical relational system, it would be impossible to define a scale.

2.1 Orders and Scale

In literature relating to measurement theory, role is extensively used to describe properties of relations [9]. Type of order can be reflexive, symmetric, transitive, complete and strongly complete etc. A quasi – order relation as described by equation:

$$a \; W \; b \leftrightarrow (a \; P \; b \; or \; a \; I \; b)$$

defines a weak relation preference where relation P represents indifference weak preference and relation I represents indifference. Also

$$\forall \, a, b \in A \rightarrow (a = b \exists \, a \; w \; b \; or \; b \; W \; a)$$

We term it as a strongly complete weak relation. This means that $\geq$ is a weak order and $> |$ is not weak since

$$if \; a = 1 \; and \; b = 1, \sim a \; W b \; and \;\; \sim b \; W \; a.$$

Similarly partial and strict partial order is also of interest with reference to Software Engineering metrics [9][10]. Both of these do not exhibit incompleteness. Partial orders are intrinsically unsuitable for precise definitions needed in Software Engineering as they depend upon preference among multi-dimensional factors ($i = 1 \; to \; n$).

Thus

$$a \; W b \text{ iff } [\, f_i \, (a) \geq f_i (b) \; \forall \; i \;]$$

The above equation defines the partial order binary relation (A, W) whereas

$$aSb \;\; \text{iff} \; (aRb \; or \; a = b)$$

defines a strict simple order, where (A, R) is a strict partial order and (A, S) is partial order.

2. 2 Scale Types

Now we investigate the scales (A, B, μ) as derived above. To fully explain why differences exist in scale, we define admissible transformation as defined by mapping

$$g: \boldsymbol{\mu} \, (A) \rightarrow B$$

whenever $(A, B, g \circ \mu)$ is a scale. Formally, function g $\circ$ μ from A to B is a homomorphism if

$$\mu : N \rightarrow Re \;\; and \;\; \mu \, (x) = 2x,$$

then $\mu \, (x)$ is a transformation. According to above $g: \mu \, (A) \rightarrow B$ is an admissible transformation. However $g(x) = -x$, $g \circ \boldsymbol{\mu}$ is not a homomorphism, so $g: \mu \, (A) \rightarrow B$ is not an admissible transformation. These admissible transformations define the following scale types:

Nominal Scales: Ordering of such scales is not possible. However non –parametric measures such as frequency and medians are permissible but measures such as standard deviation are not permissible [11][12]. **Ordinal Scales** permit ordering. Statistics are possible on such scales [12]. In such scales, relations are defined but binary operations are not. Therefore equation (a) and equation (b) may me written as

$$A = (A, R_i)\ for\ i = 1\ to\ n$$
$$B = (B, S\_i)\ for\ i = 1\ to\ n$$

Interval Scales are stronger than both the above mentioned scales as both order and intervals are preserved through admissible transformations[11][12]. **Ratio Scales** permit transformations such as $g = ax(a > 0)$. In such scales, value 0 has a meaning, for example in Kelvin and length etc [12]. **Absolute Scales**: These scales have strict constraints with transformations of the type $g(x) = x$. LOC is an example of an absolute scale [12]. For a scale to be meaningful, it is important that the transformations applied are meaningful. In other words, we can say that for a complexity measure

$$\mu(P_1 \circ P_2) > \mu(P_1) + \mu(P_2)$$

, complexity of combination of two parts is greater than the sum of two program parts. Using an internal scale and applying transformation $(g(x) = ax + b$ to both sides, we derive:

$$a\mu(P_2) + b > a\,\mu(P_1) + b + a\,\mu(P_2) + b$$

The above relationship is not meaningful for any arbitrary b. We can say that for an internal scale, equation (10) is not meaningful. However, if we convert the scale into a ratio scale as

$$a\,\mu(P_1 \circ P_2) > a\,\mu(P_1) + a\,\mu(P_2)$$

, it will become meaningful. This suggests that ratio scales are more meaningful for measuring software complexity. However extensive structures of commutativity and monotonicity are needed for a meaningful use of ratio scale.

2.3 Extensive Structures and Atomic Modifications Extensive structures and Atomic Modifications will allow for a more accurate evaluation of software complexity metrics. Any relational structure $(P, \bullet \geq, \circ)$ is an extensive structure where P is a non empty set and $\bullet \geq$ and $\circ$ depicting binary relation and binary operator respectively.

There exists no empirical relational system for software complexity evaluation. The relation at best can only be termed as empirical sub-relation as it does not contain a weak order and only contain partial information about empirical relation. Atomic modifications can be used to determine if a particular measure shod be accepted or rejected [13][14]. . They represent any prescribed changes such as addition and deletion of nodes of a DAG in calculation of cyclomatic complexity . Similarly any modification in the LOC is an atomic modification in calculation of size complexity [15][16]. However in case of measures such as cyclomatic complexity V (G) of a DAG, such modifications may be of three types: Mod1 : Addition of a node and an edge at any arbitrary location . Mod 2: Shifting an edge to a different location Mod 3: Inserting a new edge between two nodes, $V(G) = e - n + 2$. Modification 1 and Modification 2 above have no effect on the value of $V(G)$ as opposed to Modification 3 which results in an increase in the value of V(G). This also conforms to intuitive deficiencies of complexity measures as represented by empirical relational system. Hence we can deduce that if atomic modifications do not conform to empirical relational system, then the corresponding measure should not be accepted.

3. Axiomatic Approach to Metric Evaluation

With the background developed, in this section we evaluate the appropriateness of extensively used conventional Software Engineering metrics. We will adapt an axiomatic approach to evaluate these metrics formally. We present nine such axioms against which software complexity metrics can be evaluated formally. First we will briefly describe the axioms and in the next section we will present our criticism of these axioms.

We have already discussed the sensitiveness and inherent bias of measurement theory. One important property of a measure is that the produced value of complexity for a component of a program should not be more than that of the program that it is part of. In other words, a component cannot be more complex than the program that it is part of. Another important property of a good measure relates to interaction between two programs. It implies that given three programs ProgX, ProgY and ProgZ, where Prog X is called from both ProgY and ProgZ, then the interaction of ProgY and ProgX will exhibit different properties than those between Prog Z and ProX. In other words the complexity of concatenation ProX-ProgY will be different from complexity of concatenation ProgX-ProgZ. The third property states that any measure should consider statement order within a program. The fourth property states that a metric should not be affected if variables in a program are renamed. According to the last property, the sum pf complexities of components of a program should not be greater than the complexity of the program. The

basic drawback in these properties of measures which render them unsuitable for any Object Oriented paradigm, whether class based or class-less is that these are formulated not in terms of objects but in terms of programs. In a more formal representation, for any program P with complexity $|P| > 0$ and a program Q with complexity $|Q| > 0$, either $|P| \geq |Q|$ or $|Q| \geq |P|$ Furthermore if

$$|P| \geq |Q| \; and \; |Q| \geq |R|,$$

Then $|P| \geq |R|$.

This implies transitivity and weak ordering which, as discussed above makes a measure unsuitable to quantify. We will now discuss these axiomatic properties in more detail and offer our criticism of the same:

Property 1: This property formally states:

$$(\exists P)(\exists Q)(|P| \neq |Q|)$$

It ensures that any complexity metric which assigns the same complexity value to all programs or program components should not be considered

Property 2: Let c be a non negative number. Then there are only a finite number of programs of this complexity $'c'$. Property 2, ensuring sufficient resolution, is counter balanced by Property 3 which requires that metric should not be so fine that any specific value of the metric is only realized by a single program. In other word, it should be possible to find two programs that are "equally complex".

Property 3: For two distinct programs P andQ, it is possible to have $|P| = |Q|$. In our opinion, the above three properties cannot be termed as measures but properties of measures and do not reflect the syntactic or semantic nature of software complexity measure. *Property 4*: If two program components P and Q produce identical outputs for identical inputs, we can state that they have identical behavior i.e., $P \equiv Q$.

Formally

$$(\exists P)(\exists Q)(P \equiv Q \;\&\; |P| \neq |Q|)$$

Property 4 suggests that same semantics may be implemented differently syntactically in two programs. In object oriented systems, two different classes having the same message signatures, should return identical signatures. *Property 5* expresses the notion of monotonicity. It states that complexity of two concatenated (; operator) program components should be greater than the complexity of each of the component:

$$(\forall P)(\forall Q)(|P|) \leq |P;Q| \; and \; |Q| \leq |P;Q|)$$

In physical context, the property may hold true. The psychological complexity, as determined by understanding, for the whole may however be less than that of individual components.

Property 6: If a program component R is concatenated separately with components P and Q, the resulting complexity of each would not be identical:

$$6(a): (\exists P)(\exists Q)(\exists R)(|P| = |Q| \& |P;R| \neq |Q;R|)$$
$$6(b): (\exists P)(\exists Q)(\exists R)(|P| = |Q| \& |R;P| \neq |R;Q|)$$

One factor that may result in different complexities is the ordering of statements, for example the depth of nesting loops in case of functions and procedures.

Property 7: Program components P and Q exist such that Q may entirely be constituted by permutating the order of statements of P such that $|P| \neq |Q|$.

Property 8: According to this property, relabeling of individual variables or objects has no effect on complexity. Simply if P is a renaming of Q, then $|P| = |Q|$. In our opinion, this may be true for some physical measure such as LOC. In context of psychological complexity however, renaming variables can make a program virtually indecipherable.

Property 9: The final property states that complexity of a program that is constituted by concatenation of two program components may be greater than the complexity of individual components such that

$$(\exists P)(\exists Q)(|P| + |Q|) < |P;Q|)$$

In the table below, we evaluate four traditional structured metrics against these nine axiomatic properties and show that several of these axiomatic properties are not satisfied:

4. Discussion and Conclusion

In our opinion the in inability of the above axioms to satisfy traditional complexity metric stem for the fact that "complexity" as defined in these axioms is sill defined.

Table 1: Evaluation of four traditional metrics against nine axiomatic properties

Axiomatic Property	Statement Count	Cyclomatic Number	Effort Measure	Data Flow Complexity
1	Y	Y	Y	Y
2	Y	N	Y	N
3	Y	Y	Y	Y
4	Y	Y	Y	Y
5	Y	Y	N	N
6	N	N	Y	Y
7	N	N	N	Y
8	Y	Y	Y	Y
9	N	N	Y	Y

"Complexity" here refers to ease of implementation, size, and maintainability and so on. Also Property 5 and Property 6, as observed above are mutually incompatible. Property 5 is appropriate for physical complexity measures is such as size but

not suitable for measuring psychological complexity such as comprehension. Conversely Property 6, although appropriate for measuring degree of comprehension, is in appropriate for measures relating to size. Moreover, as observed above, these axioms are not consistent and not in accordance with measurement theory. For example, take the case of Property 6, Property 7 and Property 9. Property 9, although relevant on a ratio scale, is not meaningful on an internal scale whereas Property 6 and Property 7 totally reject the ratio scale.

We will now discuss each of these axiomatic properties and outline the following observations: Property 1: Cannot be applied to all software measures. For example, the measure of KNOTS, although meaningful for measuring unstructuredness of flow graphs, renders useless for structured programs as it always results in a zero value thus violating Property 1 for structured programs. Thus KNOTS, although invalid for structured programs cannot be dismissed for structured programs. Property2: Property 2 does not satisfy condition for cyclomatic complexity because of its different properties with respect to ordinal scale. It does not mean that this property is unsuitable to measure cyclomatic complexity. Property 3 warrants not criticism as it satisfies all the four conventional Software complexity metrics. Property 4 is applicable for measuring understanding and maintainability of structured programs. It is inapplicable for object oriented systems which have a high level design and implementation details at many stages of design. They may however be applicable for OO maintenance metrics. Property 5: According to this property, as discussed in extensive structures above, complexity should increase by adding new code or program component. In our opinion, in context of psychological complexity, an incomplete loop is more complex than a complete loop. This can be extended to the notion that a complete class is more understandable than an incomplete class. We however dispute applicability of this property into Object Oriented Systems form a comprehension point of view. Property 6: Absence of monotonicity in this property prevents it from becoming a ratio scale. This renders it useful for some measures and goals but not useful globally. Property 7: Absence of axiom of commutativity that is required for extensive structure makes ratio scale unattainable. Property 8: Suffices for the four conventional software complexity measures and hence no critical discussion required. Property 9: The final property is applicable for an internal scale but not for a rational scale. Formally

$$(\forall P)(\forall Q)(|P| + |Q|) < |P; Q|)$$

From the above discussion we suggest that any useful validated measure should observe the following three principles:
i)A zero complexity should result in case of an empty set.
ii) Complexity of a component of a program should always be less than the complexity of the whole program.

Table 2: Evaluation of appropriateness of the axiomatic property for complexity measures

Axiomatic Properties	Appropriate Measures	Inappropriate Measures
1	Structured Programs	Measures of unstructuredness
2	All Programs	NA
3	Size related	Control for complexity
4	Comprehension based	NA
5	Size related	Comprehension based
6	Comprehension based	Size related ; ratio scale
7	Psychological complexity	Extensive structure; ratio scale
8	Size	Comprehension related
9	Ratio scale	Ordinal Scale

iii) Measurement of a set of components of a program must produce a value less than or equal to the use of individual values produced when measuring the individual components.

The above three principles can be formally expressed through the following equations;

$$m(begin\ S_i, S_2, \ldots S_n\ end) \geq \Sigma m(S_i) \quad (a)$$

$$2(m(S_1) + m(S_2)) \geq m(if\ P\ then\ S_i\ else\ S_2) > m(S_i) + m(S_2) \quad (b)$$

$$2m(S_1) \geq m(while\ P\ do\ S) > m(S) \quad (c)$$

Applicability in Prototype (Class-less) and Class based OO Metrics

Although the requirements of both Class –less and Class based OO complexity metrics are different, our proposed properties are reasonably applicable to these. Property 1 and Property 2 are applicable as such. Property 3 in context of OO

approach implies that two classes can have the same complexity. Property 5, applied to OO systems implies that complexity of combination of two objects cannot be less than either of the component objects. Property 6 and Property 9 need further modifications to be fully applicable on OO systems. In near future , based on axioms and properties defined in this work, we will try to define a set of generic metrics that are suitable for a universal set of contexts and conditions.

References

1. Abrahamssons, P., "Commitment nets in software process improvement", Annals of Software Engineering. 2002. [2] Abrahamssons, P., Salo, O., Ronkainen, J., Warsta, J., " Agile Software development methods", Review and Analysis. VTT Publications, 2007.
2. Craig Chambers and David Ungar., "Making Pure Object-Oriented Languages Practical." In OOPSLA '91 Conference Proceedings, pp. 1-15, Phoenix, AZ, October, 1991.
3. Ole Agesen, Jens Palsberg, and Michael I. Schwartzbach. "Type Inference of Self: analysis of Objects with Dynamic and Multiple Inheritance", in Proc. ECOOP '93, pp. 247-267. Kaiserslautem,
4. Bob Hunter, Martin Fowler, & Gregor Hohpe, accessed February 12, 2013. http://www.thoughtworks.com/us/library/agile EAIMethods.pdf
5. S.R. Chidamber and C. F. Kemerer, "Towards a metrics suite for object-oriented design", in *Proc. 6th*
6. *OOPSLA Conference*, ACM 1991, pp. 197-211.
7. S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object-Oriented Design", *IEEE Trans. Software Eng.*, vol. 20, no. 6, June 1994, pp. 476-493.
8. N. E. Fenton, "Software Measurement: A necessary scientific basis", *IEEE Trans. Software Eng.*, vol. 20, no. 3, March 1994, pp. 199-206.
9. Robillard, M., Murphy, G. "FEAT: A Tool for Locating, Describing, and Analyzing Concerns in Source Code". Proc. of ICSE'03, Portland, May 2003, pp. 822-824.
10. Zacaria, A., Hosny, H. "Metrics for Aspect-Oriented Software Design". Proc. Third International Workshop on Aspect-Oriented Modeling, AOSD'03, 2003.
11. Zhao, J. "Towards a Metrics Suite for Aspect-Oriented Software". Technical-Report SE-136-25, Information Processing Society of Japan (IPSJ), March 2002.
12. Bandi, R., Vaishnavi, V., Turk, D. "Predicting Maintenance Performance Using ObjectOriented Design Complexity Metrics". IEEE Transactions on Software Engineering, 29(1), January 2003, pp. 77-87.
13. Basili, V., Caldiera, G., Rombach, H. "The Goal Question Metric Approach".
14. "An Empirical Study of the Evolution of an Agile-Developed Software System". ICSE '07 Proceedings of the 29th international conference on Software Engineering. Pages 511-518
15. S. "Theoretical reflections on agile development methodologies". Communications of the ACM - Emergency response information systems: emerging trends and technologies. CACM Homepage archive Volume 50 Issue 3, March 2007. Pages 79-83
16. Hector M. Olague. " An empirical validation of object-oriented class complexity metrics and their ability to predict error-prone classes in highly iterative, or agile, software: a case study". Journal of Software Maintenance and Evolution: Research and Practice Volume 20, Issue 3. Sept 2008.Pages 171–197,

3/5/2014