A New Suite of Metrics for Object-Oriented Software

Cara Stein[1], Letha Etzkorn[1], Glenn Cox[1], Phillip Farrington[2], Sampson Gholston[2], Dawn Utley[2], and Julie Fortune[2]

[1] Computer Science Department
[2] Industrial and Systems Engineering and Engineering Management Department
University of Alabama in Huntsville, Huntsville, AL 35899 USA

Software practitioners need to be able to assess the quality of their software, and metrics provide an automated way to do that. Traditional software metrics count aspects of code related to its syntax. In contrast, semantic metrics, introduced by Etzkorn and Delugach, count things related to the meaning of software in its domain. Because semantic metrics do not depend on the structure of the code, they can be calculated from requirements and design documents before implementation. This paper introduces and validates a suite of new semantic metrics. These metrics measure aspects of the quality of object-oriented software and can be computed from source code or from design specifications.

1 Introduction

Software practitioners need ways to assess the quality of their software. Assessment may be performed with an eye toward reuse, process improvement, or fault-proneness of the software. Metrics provide an automated way to assess software quality.

Traditional software metrics count aspects of code related to its syntax. In contrast, semantic metrics, introduced by Etzkorn and Delugach [1], count things related to the meaning of the software in its domain. As Etzkorn, Gholston, and Hughes pointed out [2], syntactic metrics are subject to distortion due to programmer style. One advantage of semantic metrics is that coding style is less likely to skew the metric values. For example, using the lines of code metric, a classic syntactic metric, the C++ code samples in Figs. 1 and 2 result in very different metric values, even though they do exactly the same thing. Although the lines of code metric is scorned by many, Curtis and Carlton [3] pointed out that most complexity metrics actually measure size and perform no better than lines of code.

```
if (balance < withdrawal) {
   bounce = true;
}
else {
   bounce = false;
}
```

Fig. 1. Code sample 1

```
bounce = (balance < withdrawal)? true : false;
```

Fig. 2. Code sample 2

If we count physical lines of source code, as recommended by [4], the first sample contains six lines of code, whereas the second contains only one. Using these values, one might conclude that the first sample accomplished more or was more complex, when in reality they both accomplish the same task. Moreover, the first code sample is easier for a non-C++ expert to understand, so it may be easier to maintain.

In contrast, semantic metrics are concerned only with the ideas involved. In this example, semantic metrics would pick out the concepts of balance, withdrawal, and bounce in the banking domain, without regard to the way the code was written. Thus differences in programmer style and even programming language used will not impact semantic measures of the system's size or complexity.

Furthermore, because semantic metrics do not depend on the structure of the code, they can be calculated from requirements and design documents before the code has been written [2]. This allows quality to be assessed earlier in the development process, when changes are cheaper to make.

2 Validating Metrics

Kitchenham, Pfleeger, and Fenton [5] proposed a framework for validating measurements of software. First, they identified necessary concepts for measurement:

- Entities – things we want to know more about
- Attributes – properties we want to measure
- Units – mapping between an attribute and number system (ex. inches)
- Scale types – nominal, ordinal, interval, or ratio

Thus, a metric value should measure an attribute of an entity using some unit. Units are valid for ratio and interval data and can be adapted to work with ordinal data [5].

The four aspects of validity for a measure are:

- Attribute validity: the entity being studied has the attribute.
- Unit validity: the unit used is appropriate for the attribute.
- Instrumental validity: the underlying model is valid and the measurement instrument is calibrated.
- Protocol validity: the measurement is taken in a way that is consistent, unambiguous, and prevents problems such as double counting [5].

Generally, there are direct measures and indirect measures. For direct measures, the following theoretical properties should apply:

- There exist at least two different entities that have different values for the attribute.
- The measurement must work in a way that makes sense with respect to human understanding of the attribute.
- If the attribute is part of a valid measurement, any of its units can be used.

- It is possible for different entities to have the same attribute value [5].

For indirect measures, once the model on which they are based is shown to be correct, they must also make appropriate use of units and scales, and they must not have discontinuities such as division by zero [5].

More specifically, Briand, Morasca, and Basili [6] proposed a set of criteria for different types of software metrics.

Complexity:

- Nonnegativity: the value is never negative
- Null Value: the value is zero if there are no relationships between parts of the entity being measured
- Symmetry: the direction of relationships does not impact the value
- Module Monotonicity: the complexity of a system is greater than or equal to the sum of the complexities of any set of disjoint modules
- Disjoint Module Additivity: the complexity of a system of modules with no elements in common is the sum of the complexities of the modules [6]

Cohesion:

- Nonnegativity: the value is never negative
- Normalization: the value always falls within a specified range, so that systems of different sizes can be compared
- Null Value: the value is zero if there are no relationships within a module or system
- Monotonicity: the value never decreases as a result of adding relationships within a module
- Cohesive Modules: the value never increases as a result of merging unrelated modules [6]

Coupling:

- Nonnegativity: the value is never negative
- Null Value: the value is zero if there are no relationships between modules in a system
- Monotonicity: the value never decreases as a result of adding relationships between modules
- Merging of Disjoint Modules: the value never increases as a result of merging disjoint modules [6]

Two other aspects of software that can be measured are overlap of classes or member functions, and key module status. Key module metrics indicate the relative importance of a class to a system or of a function to a class. We propose the following properties for valid measures of overlap and key module status.

Overlap:

- Nonnegativity: the value is never negative
- Null Value: the value is zero if there is no common functionality between modules
- Monotonicity: adding functionality to a module never decreases the value
- Merging Modules: the value never increases as a result of merging modules with no common functionality

Key Module Status:

- Nonnegativity: the value is never negative
- Null Value: the value is zero if the module performs no functionality
- Monotonicity: adding functionality to a module never decreases the value
- Merging Modules: if two modules are merged, the resulting module has a value at least as large as each of the values for the two modules before they were merged.

3 Proposed Semantic Metrics

Most of the original semantic metrics defined by Etzkorn and Delugach [1] rely on a knowledge base that has a strong conceptual graph structure. Conceptual graphs are one format for expressing information and the relationships between concepts. However, not all knowledge bases are built using conceptual graphs.

New semantic metrics that do not depend upon a particular knowledge base structure are needed. This work is especially needed in the area of cohesion, because all of the current semantic cohesion metrics rely heavily on a conceptual graph structure in the knowledge base. The metrics proposed in this paper can be used with any knowledge base that can associate ideas with classes and class members.

To define the proposed metrics, we need a few background definitions. Let $\{C_1, C_2, \ldots, C_m\}$ be the set of classes in a system. Let $\{M_{a1}, M_{a2}, \ldots, M_{an}\}$ be the set of the n member functions of class C_a. Let an idea be a concept or keyword in the knowledge base. For each member function M_{ai}, let I_{ai} be the set of all ideas associated with that function. For each class C_a, let I_a be the set of all ideas associated with that class. Let ~ be a relation on the set of ideas in the knowledge base, such that a~b if there exists a conceptual relation from a to b, there exists a conceptual relation from b to a, there exists an inference relation from a to b, or there exists an inference relation from b to a.

All of the metrics proposed in this paper are indirect measurements based on the direct measurement that results from counting concepts and keywords belonging to a class or member function. That is, given a listing of all of the ideas in a knowledge base and a list of classes and members with which each idea is associated, the sets $\{I_1, I_2, \ldots, I_m\}$ and $\{I_{a1}, I_{a2}, \ldots, I_{an}\}$ are generated. The cardinality of each of these sets is the measure of semantic mass of the corresponding class or member function.

To evaluate this measure using the criteria suggested by Kitchenham, Pfleeger, and Fenton [5], first we must identify what we are measuring. Here, the entities are classes and member functions, the attribute is semantic mass, the unit is the idea, and the scale of the values is ratio. The measure meets the four aspects suggested in [5] for valid measures as follows:

- Attribute validity: because semantic mass is a measure of how many ideas are associated with a class or member, clearly the entities of class and member have the attribute of semantic mass.
- Unit validity: idea is an appropriate unit for semantic mass.
- Instrumental validity: the instrument is valid if the knowledge base associates a class or member with the ideas within the domain that relate to the functionality of the class or member.
- Protocol validity: the measurement as described above using set theory is unambiguous, consistent, and prevents double counting.

Based on this measurement, we define new semantic metrics that are calculated for each class.

3.1 Cohesion Metrics

Cohesion is the degree to which the functionality of a software module all goes together or works toward a common goal. If a class is cohesive, there will be some commonality of concepts and keywords among its functions. For purposes of the Briand, Morasca, and Basili criteria for cohesion metrics [6], we define a relationship between modules to be an idea common to both modules.

LDM (Logical Disparity of Members). For each pair of member functions in the class, count all of the ideas that are not shared. That is, count the ideas belonging to one function in the pair but not the other and divide that by the number of unique ideas belonging to either function in the pair. Add those up and divide by the number of pairs of functions. (This is a measure of lack of cohesion.) LDM for class C_a is computed by equation 1.

$$x_{ij} = \frac{\left|\{y \mid y \in I_{ai} \wedge y \notin I_{aj}\}\right|}{\left|\{z \mid z \in I_{ai} \vee z \in I_{aj}\}\right|}, \quad \textbf{(1)}$$

or 0 if $I_{ai} \cup I_{aj} = \varnothing$

$$\text{LDM} = \frac{\sum_{i=1}^{n} \sum_{j=1}^{n} x_{ij}}{\frac{n(n-1)}{2}},$$

or 0 if $n < 2$.

Evaluation: The value of this metric is never negative and always falls in the range [0,1]. Because this is a measure of lack of cohesion, the usual criterion of having a value of zero when there are no shared ideas is inappropriate. Instead, LDM has a value of zero when there are no unshared ideas. LDM also has a value of zero in the trivial case where no member functions have any associated ideas, but we feel this is appropriate given the definition of the metric: if no functions have any ideas, there is no logical disparity among them. Similarly, because LDM is a measure of lack of cohesion, its value never increases as a result of adding related modules within a class, and its value never decreases as a result of merging unrelated modules.

PSI (percentage of shared ideas). PSI is the number of ideas shared by at least two member functions of a class, divided by the number of ideas associated with any member function in the class. PSI for class C_a is computed by equation 2.

$$\text{PSI}=\frac{\left|\{x \mid \exists i,j : x \in I_{ai} \wedge x \in I_{aj}\}\right|}{\left|\{y \mid \exists k : y \in I_{ak}\}\right|} \text{ for } 1 \le i, j, k \le n, \tag{2}$$

or 0 if no ideas are associated with any function of the class.
Evaluation: The value is never negative and always falls in the range [0,1]. The value is zero if there are no common ideas between any member functions in the class. The value never decreases as a result of adding common ideas between functions, and the value never increases as a result of merging unrelated functions.

PUI (percentage of universal ideas). PUI is the number of ideas shared by all member functions of a class, divided by the number of ideas associated with any member function in the class. PUI for class C_a is computed by equation 3.

$$\text{PUI}=\frac{\left|\{x \mid \forall i, x \in I_{ai}\}\right|}{\left|\{y \mid \exists k : y \in I_{ak}\}\right|} \text{ for } 1 \le i, k \le n, \tag{3}$$

or 0 if no ideas are associated with any function of the class.
Evaluation: The value is never negative and always falls in the range [0,1]. As with PSI, PUI also fulfills the properties of null value, monotonicity and cohesive modules.

LORM2b (logical relatedness of methods 2b). LORM2b is a variation on Etzkorn and Delugach's LORM2 [1]. LORM2b performs the same calculation as LORM2 except that LORM2b counts keywords as well as concepts. LORM2b for class C_a is computed by equation 4.

$$x_{ij} = \frac{2 * \left|\{y \mid y \in I_{ai} \wedge y \in I_{aj}\}\right|}{|I_{ai}| + |I_{aj}|}, \quad (4)$$

or 0 if $I_{ai} \cup I_{aj} = \varnothing$;

$$\text{LORM2b}=\frac{\sum_{i=1}^{n}\sum_{j=i+1}^{n} x_{ij}}{\frac{n(n-1)}{2}},$$

or 0 if n < 2.

Evaluation: The value is never negative and always falls in the range [0,1]. The value is zero if no ideas are held in common between any functions. The value never decreases as a result of adding common ideas to functions nor increases as a result of merging unrelated functions.

3.2 Overlap Metrics

Class overlap metrics measure the degree of overlap between the functionality of two classes. If two classes have many ideas in common, one can assume that there is overlap in their functionality.

PCRC (proportion of closely related classes). PCRC is the number of classes with which this class shares over half of its ideas, divided by the number of classes in the system. PCRC for class C_a is computed by equation 5.

$$\text{PCRC}=\frac{\left|\left\{C_i \mid \left(\left|\{x \mid x \in I_i \wedge x \in I_a\}\right| > \frac{|I_a|}{2}\right)\right\}\right|}{m-1} \quad (5)$$

for $1 \le i \le m$, $i \neq a$,

or 0 if m=0.

Evaluation: The value is never negative, and it is zero if there are no common ideas between the class being evaluated and other classes. If functionality is added, it will either add one or more classes to the set of C_i above or it will do nothing to the set. Therefore, adding functionality can never decrease the value of PCRC. Also, the value never increases as a result of merging disjoint functions. However, PCRC does not meet the merging modules property. For example, if a system consists of classes C_a, C_x, and C_y such that $I_a = \{i_1, i_2, i_3, i_4, i_5, i_6, i_7\}$, $I_x=\{i_1, i_2, i_3\}$, and $I_y=\{i_4, i_5, i_6\}$. PCRC(C_a)=0 because no other class shares over half of C_a's ideas. However, if disjoint classes C_x and C_y are merged, the resulting class does share over half of C_a's ideas, so PCRC(C_a)=1.

APISOC (average proportion of ideas shared with other classes). APISOC is the average of the proportion of ideas shared with each other class in the system divided by the total number of unique ideas in that pair. APISOC for class C_a is computed by equation 6.

$$x_i = \frac{|\{y \mid y \in I_i \wedge y \in I_a\}|}{|\{z \mid z \in I_i \vee z \in I_a\}|}, \tag{6}$$

or 0 if $I_i \cup I_a = \varnothing$;

$$\text{APISOC} = \frac{\sum_{i=1}^{m} x_i - 1}{m-1},$$

or 0 if $m < 2$ or $|I_a|=0$.

Evaluation: The subtraction of one in the numerator may appear to introduce the possibility of a negative value, but the one being subtracted is the overlap of the class with itself. If this value is not one, it can only be zero, in which case the whole value of APISOC is zero. Therefore, the value of APISOC is never negative. Similarly, if a class has no common functionality with any other classes in the system, its value will be 1-1=0. APISOC also meets the requirements of monotonicity and merging modules.

3.3 Complexity Metrics

The more concepts and keywords a class contains, the more complex its role is with respect to the application domain.

CDC2 (class domain complexity 2). CDC2 is a variation on Etzkorn and Delugach's CDC [1]. CDC2 performs the same calculation as CDC except that CDC2 includes keywords as well as concepts and conceptual relations. CDC2 for class C_a is computed by equation 7.

$$\text{CDC2} = \sum_{i \in I_a} \left(1 + |\{x \in I_a \mid x \sim i \wedge x \neq i\}| * w_i\right) \tag{7}$$

where w_i is a weighting factor for the complexity of an idea on the following scale:

1.0 = complex, 0.5 = average, and 0.25 = simple.

Evaluation: The value is never negative, and the direction of relationships does not impact the value because ~ is symmetric by definition. If the CDC2 calculation is performed on a system level rather than a class level, its value is at least the sum of the CDC2 values for disjoint classes. However, the value is not zero unless a class has no ideas associated with it, so CDC2 does not meet the Null Value criterion. If CDC2 is calculated on a system level rather than a class level, it also fails to have the property of Disjoint Module Additivity, because disjoint classes C_a and C_b could have ideas x and y such that $x \neq y$ but $x \sim y$. In that case, the relationship would increase the system-wide CDC2 value to more than the sum of the CDC2 values for the classes.

RCDC2 (relative class domain complexity 2). RCDC2 is a variation on Etzkorn and Delugach's RCDC [1]. RCDC2 performs the same calculation as RCDC except that RCDC2 includes keywords as well as concepts and conceptual relations. RCDC2 for class C_a is computed by equation 8.

$$\text{RCDC2} = \text{CDC2}(C_a) / \max(\text{CDC2}(C_i)) \text{ for } 1 \leq i \leq m, \text{ or 0 if all CDC2 values in the system are 0.} \quad (8)$$

Evaluation: The value is never negative, and the direction of relationships does not impact the value. However, none of the other criteria for complexity measures applies to RCDC2, because it is a measure of relative complexity, not complexity.

3.4 Key Class Metrics

If a class touches on or contains most of the ideas of the system, then that class must perform considerable or important functionality. Such a class may be considered a key class of the system.

KCF (key class factor). KCF is the number of ideas belonging to the class divided by the number of unique ideas belonging to any class in the system. KCF for class C_a is computed by equation 9.

$$\text{KCF} = \frac{|I_a|}{|\{y \mid \exists k : y \in I_k\}|} \text{ for } 1 \leq k \leq m, \quad (9)$$

or 0 if no class has any ideas associated with it.

Evaluation: The value is never negative, and the value is zero for a class that performs no functionality, because such a class would have no ideas associated with it. The value could never decrease as a result of merging modules, because merging modules would cause the numerator to increase or stay the same while not impacting the denominator. KCF also satisfies the Merging Modules property, because the KCF of a merged class is at least the KCF value of each of the classes before they were merged.

KCI2 (key class indicator 2). KCI2 is a variation on Etzkorn and Delugach's KCI [1]. KCI2 performs the same calculation as KCI except that KCI2 includes keywords as well as concepts and conceptual relations in the calculation. KCI2 for class C_a is computed by equation 10.

$$\text{KCI2} = 1, \text{ if RCDC2}(C_a) \geq .75, \text{ 0 otherwise.} \quad (10)$$

Evaluation: KCI2 is a nominal measure, so most validation criteria do not apply. However, it does meet the Nonnegativity and Symmetry properties of key module measures.

4 Conclusion

Semantic metrics provide a mechanism for assessing the quality of software in the design or implementation phases. While assessing software in the implementation phase is important, being able to assess earlier provides added value in the ability to correct mistakes or potential problems earlier in the software development lifecycle, when changes are less expensive to make. Whereas most of the previous semantic metrics required a conceptual graph-based knowledge base, the proposed set of metrics can be calculated using any knowledge base that associates classes with ideas. Thus, the proposed set of metrics can be computed using a broader range of knowledge bases than could be used with previously existing semantic metrics.

6 Acknowledgements

The research in this paper was partially supported by NASA grants NAG5-12725 and NCC8-200.

References

1. Etzkorn, L., Delugach, H.: Towards a Semantic Metrics Suite for Object-Oriented Design. Proceedings of the 34th International Conference on Technology of Object-Oriented Languages and Systems (2000) 71-80.
2. Etzkorn, L., Gholston, S., Hughes, W.: A Semantic Entropy Metric. Journal of Software Maintenance and Evolution, Vol. 14, No. 4 (July/August 2002) 293-310.
3. Curtis, B., Carleton, A.: Seven ± Two Software Measurement Conundrums. Proceedings of the 2nd International Metrics Symposium (1994) 96-105.
4. Park, R.: Software Size Measurement: A framework for Counting Source Statements. Technical Report SEI-92-TR-20. Software Engineering Institute, Pittsburgh (1992) 136-137.
5. Kitchenham, B., Pfleeger, S., Fenton, N.: Towards a Framework for Software Measurement Validation. IEEE Transactions on Software Engineering, Vol. 21, No. 12 (Dec. 1995) 929-944.
6. Briand, L., Morasca, S., Basili, V.: Property-Based Software Engineering Measurement. IEEE Transactions on Software Engineering, Vol. 22, No. 1 (Jan. 1996) 68-86.