Towards a Semantic Metrics Suite for Object-Oriented Design

Letha Etzkorn
Computer Science Department
Huntsville, AL 35899 USA
letzkorn@cs.uah.edu

Harry Delugach
Computer Science Department
Huntsville, AL 35899 USA
delugach@cs.uah.edu

Abstract

In recent years much work has been performed in developing suites of metrics that are targeted for object-oriented software, rather than functionally-oriented software. This is necessary since good object-oriented software has several characteristics, such as inheritance and polymorphism, that are not usually present in functionally-oriented software. However, all of these object-oriented metrics suites have been defined using only syntactic aspects of object-oriented software; indeed, the earlier functionally-oriented metrics were also calculated using only syntactic information. All syntactically-oriented metrics have the problem that the mapping from the metric to the quality the metric purports to measure, such as the software quality factor "cohesion," is indirect, and often arguable. Thus, a substantial amount of research effort goes into proving that these syntactically-oriented metrics actually do measure their associated quality factors.

This paper introduces a new suite of semantically-derived object-oriented metrics, which provide a more direct mapping from the metric to its associated quality factor than is possible using syntactic metrics. These semantically-derived metrics are calculated using knowledge-based, program understanding, and natural language processing techniques.

Keywords
object-oriented metrics, program understanding, natural language processing, knowledge-based systems, semantic networks, conceptual graphs

1. Introduction

In the last several years, various object-oriented metrics suites have been developed [2,3,6,10,19,21,22,23,24]. Although many of these metrics have been successful in varying degrees [4,17]; still, since all of these object-oriented metrics suites are derived from syntactic aspects of object-oriented code, the mapping from each metric to the quality factor that it purports to measure is arguable. Henderson-Sellers notices this aspect of the usual syntactic cohesion measures, when he says [19]:
"It is, after all, possible to have a class with high internal, syntactic cohesion but little semantic cohesion."

Henderson-Sellers gives the example of a class that includes features of both a person and the car that the person owns [19]. Assuming that there is a one-to-one correspondence between each person and his/her car (each person may own one car, each car belongs to one and only one person), then a class named CAR_PERSON is possible. This class could be syntactically and internally highly cohesive as measured by, for example, the Lack of Cohesion

0-7695-0774-3/00 $10.00 © 2000 IEEE

in Methods (LCOM) metric [6,17,19]. However, from a semantic standpoint, as a whole class seen from outside the system, the combination of a car with a person in a single class is not a sensible class definition [19]. Thus, in this case the aspect of cohesion measured by the syntactic cohesion metrics such as LCOM is not a good measure of the high level cohesion of the class. This occurs since there is not a direct mapping from the LCOM metric to the software quality factor of "cohesion".

Many of the software quality factors that syntactic software metrics attempt to indirectly measure can be measured more directly by use of an equivalent semantic metric. The definition and application of these semantic metrics is possible by the use of knowledge-based, program understanding, and natural language processing techniques. This paper presents a suite of object-oriented semantic metrics, and compares these metrics to earlier syntactic metrics suites. The semantic metrics are evaluated using Weyucker's properties [26] and Briand et al.'s cohesion properties [5] as appropriate.

2. Syntactic metrics suites for OO design

The Chidamber and Kemerer (also referred to in this article as C&K) syntactic metrics suite consists of six metrics [6]:

1. Depth of Inheritance Tree (DIT)—This metric measures how many classes can affect the current class through inheritance.
2. Number of Children (NOC)—This metric measures the scope of the influence of a class on its subclasses due to inheritance.
3. Response For a Class (RFC)—This metric counts the set of methods that can potentially be executed in response to a message received by an object of the current class.
4. Lack of Cohesion of Methods (LCOM)—The cohesion of a class is characterized by how closely the local methods are related to the local attributes.
5. Weighted Methods per Class (WMC)—The complexity of a class is the sum of the complexity of a class' local methods.
6. Coupling Between Objects (CBO)—This metric is a count of the number of other classes to which the current class is coupled, via non-inheritance-related couples. Two classes are coupled when the methods of one class use methods or attributes of another class.

Li developed another Object-Oriented syntactic metrics suite that addressed certain shortcomings in Chidamber and Kemerer's metrics suite[21,23]:

1. Number of Ancestor Classes (NAC)—This metric measures the total number of ancestor classes from which a class inherits in the class inheritance hierarchy. It addresses a problem with multiple inheritance in the C&K DIT metric.
2. Number of Descendant Classes (NDC)—This metric measures the number of classes that may potentially be influenced by the class because of inheritance relations. It addresses a problem with the C&K NOC metric, in which the C&K NOC metric counted only the immediate children of a class, and not the grandchildren. A class influences all its subclasses and not just the immediate children.
3. Number of Local Methods (NLM)—This metric counts the number of local methods defined in a class which are accessible outside the class. Li felt that this metric better defines one of two possible versions of the C&K WMC metric (the other version is better defined by CMC).
4. Class Methods Complexity (CMC)—This metric is the sum of the internal structural complexity of all local methods, regardless of whether they are visible outside the class or not. Li felt that this metric better defines one of two possible versions of the C&K metric

(the other version is better defined by NLM).

5. Coupling Through Abstract Data Types (CTA)—This metric counts the total number of classes that are used as abstract data types in the data attribute declaration of a class.
6. Coupling Through Message Passing (CTM)—This metric measures the number of different messages sent out from a class to other classes, excluding the messages sent to the objects created as local objects in the local methods of the class.

Note: Li used different names for his metrics in different papers, but his papers generally employ the same set of metrics [21,22]. Li still employs a definition of the LCOM metric to measure the cohesion of a class (this LCOM definition is consistent with that of Li and Henry [23], and is different from the later definition of LCOM provided by C&K [6].

The C&K and Li metrics suites have been examined here since they are widely studied syntactically-based Object-Oriented metrics suites, particularly the C&K metrics suites. Other syntactically-based metrics suites have also been defined. [19,24].

3. Limitations of syntactic metrics suites

If the Chidamber and Kemerer metrics suite and the Li metrics suite are considered in terms of quality analysis, it is clear that these metrics suites provide metrics that attempt to indicate how a class rates in relation to the software quality factors Cohesion, Complexity, and Coupling. Cohesion is measured with LCOM (both C&K and Li), Complexity is measured with WMC (C&K) and NLM and CMC (Li), Coupling is measured with DIT and NOC (C&K), and with CTA, CTM, NAC, and NDC (Li).

The Li metric NLM can be used as a general class Complexity measure, but it also is considered as a measure of the complexity of the interface to a class. In a sense, all of the above metrics could be considered to be related to Complexity. The LCOM metric, for example, could be considered a Complexity metric when determining whether a class provides more functionality than is desired. The less cohesive the class is, the more unwanted and unneeded complexity is reused along with the needed functionality if the class is reused elsewhere than in its original system. The various Coupling metrics could be considered complexity metrics for similar reasons (adding functionality and therefore complexity to the class). However, the main thrust of the above metrics is toward the quality factors specified for each above.

In addition to Cohesion, Complexity, and Coupling, other qualities, or quality factors are also important to an object-oriented design, and to object-oriented implementation. One such quality is whether the current class is a key (core) class of the design or implementation. Key classes are central to the implementation, or business domain in which the software is being developed. They are typically discovered early in the analysis, and provide a start for estimating the total amount of work remaining in a project—the number of key classes is an indicator of the volume of work needed in order to develop an application[24]. They are also the central points of reuse on future projects, since they are highly likely to be needed in similar business or implementation domains [24]. The number of key classes is a count of identified classes that are deemed to be of central importance to the business or implementation domain in question. None of the current syntactic metrics suites provide a determination of whether a class is a key class or not.

Another quality factor that is important to the design of a class is the amount of functional overlap of one class with another class. If two classes are very heavily overlapped in functionality, then the system has probably been poorly designed, and the selection and breakdown of the system's classes and the tasks assigned to classes should be reexamined. If

no classes overlap in functionality, then it is possible that the system's chosen class structure has resulted in a structure with a high communication overhead between classes. In this situation also, the selection and breakdown of classes in the system should be reexamined. None of the current syntactic metrics suites provides a determination of the functional overlap of different classes. The LCOM metric measures the cohesion of a class by attempting in an indirect manner to measure the functional overlap of the methods of a class (rather than the overlap of the class with other classes). However, several problems with the LCOM metric have been identified. Some of these problems are discussed by Basili et al. [4], Hitz and Montazeri[20], and Etzkorn et al. [17]. Most of these problems are due to the indirect measurement technique employed: how closely the local methods are related to the local instance variables; two methods are considered related if they access the same variable. Also, this method for calculating cohesion is not easily extendable to the calculation of class overlap.

Another aspect of a software system that can be difficult to measure is the quality of its documentation in the form of comments and good identifier names. Current practice is to simply count the number of comments in a class. However, this counting of comments does not give any kind of indication as to the descriptive quality of the comment, and says nothing about the descriptive quality of identifier names.

4. A proposed semantic metrics suite for object-oriented design

We here propose a suite of automatable semantic design metrics to measure the class cohesion, class domain complexity, key class identity, class interface complexity, class overlap, and the documentation quality of a class. Our metrics are proposed within the context of knowledge-based systems for analyzing object-oriented software that contain knowledge-bases consisting of semantic networks formed from conceptual graphs. This is a good choice since an ANSI (American National Standards Institute) standard for conceptual graphs has recently been approved. Various tools are being developed to meet this standard.

One example of a knowledge-based system containing such a semantic network is a natural language-based program understanding system, called the **P**rogram **A**nalysis **T**ool for **R**euse (the **PATR**icia system), which was originally intended for the identification and qualification of reusable components in object-oriented legacy code [11,13,14,15,16]. In order to determine whether a particular class or class hierarchy is useful, and therefore potentially reusable in a given domain, the **PATR**icia system performs natural language understanding on comments and identifiers drawn from object-oriented code. Output reports include a list of concepts identified in a class or class hierarchy that are associated with the implementation-domain, with definitions, and a description of the functionality of a class in natural language.

The **PATR**icia system employs a semantic phase, with a knowledge-base which is a weighted, hierarchical semantic network. Concepts in the semantic net are stored as conceptual graphs [25] or as part of a conceptual graph.

The semantic network has an interface layer of syntactically-tagged keywords [11,14]. A keyword in the interface layer that is an adjective, for example, is a different keyword than a keyword which is a noun. Parsed comment sentences, and syntactically tagged keywords are compared by the CLIPS inferencing engine to the interface layer of the semantic net.

Both interior concepts and interface nodes allow for information regarding the location (the name of the class or classes) where that concept was identified.

The conceptual graphs within the semantic network consist of concepts and conceptual relations. After semantic analysis of an object-oriented system, each class or method analyzed by the **PATR**icia system has associated with it a set of concepts and conceptual relations.

The semantic metrics proposed below are evaluated as follows:

1) the cohesion and overlap metrics are examined within the context of the cohesion metrics properties specified by Briand et al [5]
2) All other metrics are examined in relation to the list of software metric evaluation criteria provided by Weyucker [26].

4.1 Class cohesion

We propose a metric to measure the cohesion of a class that we call **LO**gical **R**elatedness of **M**ethods (**LORM**). According to Eder et al., [5, 9], the most desirable cohesion is Model cohesion, in which the class represents a single, semantically meaningful concept. To determine whether a class represents a single, semantically meaningful concept, the LORM metric measures the conceptual relatedness of the methods of the class, as determined by the understanding of the class methods represented by a semantic network of conceptual graphs.

To collect this metric, semantic analysis by a knowledge-based (expert) system is applied to the class. Each method/member function is understood separately, and the results of the understanding for each method is stored in the conceptual graphs and concept groups of the semantic network which forms the knowledge-base of the expert system.

LORM is defined as follows:

LORM = total number of relations in the class / total number of possible relations in the class

total number of relations in the class = number of pairs of methods in the class for which one method contains conceptual relations forming external links out of the set of concepts that belong to the method (conceptual relations to other concepts belonging to the function itself are not counted) to or from the set of concepts belonging to another method in the class.

total number of possible relations = number of pairs of methods (member functions) = N choose 2 = n! / 2! (n-2)! = n (n-1) / 2

n = total number of member functions (methods) in the class.

Any concept belonging to both f_i and f_j (overlapping concept) is not included in the calculation.

The above metric can be expressed graph-theoretically as:

Let X denote a class, M_X the set of methods of the class, and C_X the concepts and conceptual relations of a class. We denote CR_X as the set of conceptual relations of a class, and CO_X as the set of concepts of a class: $CR_X \cup CO_X = C_X$. Let n denote the number of methods of the class. For each function $f_i \in M_X$, $1 \le i \le n$, we denote $C_i \in C_X$ as the set of concepts and conceptual relations of f_i. We denote $CR_i \in CR_X$, $CR_i \in C_i$ as the set of conceptual relations of f_i, and $CO_i \in CO_X$, $CO_i \in C_i$ as the set of concepts of f_i. Consider a graph G_X (V,E) with $V=M_X$, $E_i = \{< f_i, f_j > \in V \mathbf{X} V \mid \exists\ p \in CR_i$: p connects from $c_i \in CO_i$ to $c_j \in CO_j \vee$ p connects from to $c_j \in CO_j$ to $c_i \in CO_i \}$, $E = \cup_{i=1}^{m} \{ E_i \}$, $Emax = \{< f_i, f_j > \in V \mathbf{X} V \mid \forall\ f_i \in M_X, f_j \in M_X, 1 \le i \le n-1, 1 \le j \le n$, r connects from f_i to f_j, $r \in CR_i\}$. We define $|Q_i| = \{ 0$ if $|E_i| = 0$, 1 if $|E_i| \ge 1\}$. Then $LORM = (\sum_{i=1}^{n-1} |Q_i|) / |Emax|$.

When we say, "the concepts of a class," "the conceptual relations of a class," "the concepts and conceptual relations of a method," etc., we mean the concepts or conceptual relations that have been identified by semantic processing of the class (or method), as being associated in the knowledge-base with that class or method.

This metric measures the number of edges of a graph divided by the max number of possible edges in the graph. In this case, the graph vertices are methods in the class, and the edge from one vertex to another is represented as the set of conceptual relations linking the vertices. The mathematical formulation of the LORM metric is similar to that of the Co' metric defined by

Briand et al. [5]; however, the LORM metric differs from the Co' metric in that it is semantically-related, while the Co' metric is a syntactically-related metric that employs method accesses of attributes. LORM satisfies all four cohesion properties specified by Briand et al [5], except that for cohesion property 3, if additional relationships were added between two methods that had already been linked, the cohesion value would not increase.

The LORM metric determines the relatedness of the separate tasks performed by methods, but does not measure an overlap in functionality. A different measure, that determines the overlap of functionality, is to examine the concepts that appear in more than one function. We call this metric LORM2, but do not present it here due to space limitations.

Note that these metrics employ only local methods in the calculations. Whether adding employing inherited methods in the calculation would improve the cohesion calculations requires further study.

The methods of a class are supposed to be logically related (logical cohesion). However, it is not useful for the methods to have the same functionality, that is, to perform the same tasks. Thus the LORM2 metric should be used with care. A value close to 1 for this metric would be expected if all methods performed the same task! However, a value close to 0 for this metric would indicate that no methods perform any related tasks. Thus, "goodness" bounds for this metric must be empirically derived.

The LORM2 metric has a drawback in that it considers only concepts and not conceptual relations. If both are considered, then the problem becomes one of detecting overlap by performing the intersection of conceptual graphs associated with each method. This leads to another semantic cohesion metric:

Definition. For two conceptual graphs, v_1 and v_2, given concept(s) c_i, ($1 <= i <= n$, where n is the total number of concepts in v_1), and concepts d_j, ($1 <= j <= m$, where m is the total number of concepts in v_2), such that the criteria for a compatible projection hold, for $1<=q<=i$, $1 <= r <= j$:

- Type($\pi_1 c_q$) $\cap$ Type($\pi_2 d_r$) $> \tau$, where τ is the universal type
- The referents of $\pi_1 c_q$ and $\pi_2 d_r$ conform to Type($\pi_1 c_q$) $\cap$ Type($\pi_2 d_r$).
- If referent($\pi_1 c_q$) is the individual marker s, then referent ($\pi_2 d_r$) is either s or *.

Where π_1: $v_1 \rightarrow w$, π_2: $v_2 \rightarrow w$.

Then there exists a graph w that contains all concepts in the compatible projection of c_i and d_j. We define w as the *intersection* (*maximal common subgraph*) of conceptual graphs v_1 and v_2, and $|w|$ is the number of concepts plus the number of conceptual relations in w.

Thus, a definition for this cohesion metric is as follows:

$$\textbf{LORM3} = (1/M) \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} \sum_{k=1}^{Q} [\ (1/Q)\ (\ |w_{iqj}| \ / \ \min(|v_{iq}|, |v_{jq}|)\ )$$

where,

$| w_{ij} | = |\cup_{q=1}^{Q} \{w_{iqj}\}|$ = number of concepts and conceptual relations in the set of maximal common subgraphs that form the intersection set of the set of conceptual graphs belonging to method f_i and the set of conceptual graphs belonging to method f_j.

N = number of methods in the class

M = number of pairs of methods = N choose 2 = N! / 2! (N-2)! = N (N-1) / 2

Q = number of separate conceptual graphs in the method

This metric meets all four cohesion properties of Briand et al. [5]. Note that this metric is measuring concept similarity, rather than concept equality.

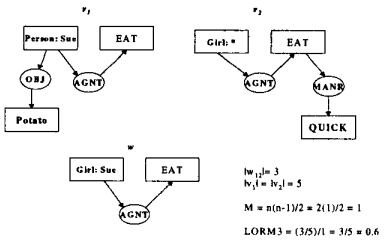

Figure 1. LORM3 Calculation

4.2 Class domain complexity

Complexity is probably the most commonly measured software quality. However, most previous object-oriented complexity metrics have been syntactically based. The closest approach to a semantic object-oriented complexity metric in the past has been the concept of entropy, originally used in the area of information theory to estimate the content of messages. This was shown to be useful in evaluating procedural software's code complexity [8, 18], and more recently was used to examine object-oriented design complexity [2].

We wish to propose two new metrics to measure the domain complexity of a class. Metrics that measure the domain complexity of a class examine a different quantity than the syntactic complexity measures of the past. The syntactic metrics measure the complexity of the program, and in those metrics it is the details of the implementation that determine the program's complexity. Metrics that measure domain complexity measure the complexity of the problem, rather than the complexity of the program. They also provide a description of the psychological complexity, or the difficulty of understanding, of the program. We call the first metric Class Domain Complexity (CDC), and the second Semantic Class Definition Entropy (SCDE). The first metric, CDC, is described here. The second, SCDE, is obtained by applying the concept of entropy to domain information. It is described in detail elsewhere [12]. Thus, we define Class Domain Complexity as follows:

CDC = $\Sigma_{i=1}^{m}$ **|concept plus its associated conceptual relations| X Weighting Factor.**

where **m** = number of concepts associated with the class

|concept plus its associated conceptual relations| = 1+number of conceptual relations linking the current concept to another concept recognized by the class. Concepts linking to concepts not recognized by the class are not included in the count. Only outgoing conceptual relations are included in the count, to prevent counting a conceptual relation multiple times.

Similarly to the usage in the Function Point metrics [1], the Weighting Factor is divided into Simple, Average, and Complex values. However, here the Weighting Factor is an indication of the abstraction level of the concept, and is stored as part of the concept when the knowledge-base is developed. Very specific simple concepts would receive a "simple" weight, while concepts that relate to large domain items would receive a "complex" weight.

Initially we define the **Weighting Factor** of the concept for the CDC metric as follows : Complex = 1.0, Average = 0.50, Simple = 0.25.

This choice of values are initial rules of thumb, and should be later verified by empirical evidence. An alternative method for determining concept complexity would be to employ the depth of the concept within the type hierarchy as an indication of complexity.

The CDC metric satisfies Weyucker's properties 1, 2, 3,and 5. CDC does not meet property 4 since it provides a measure of domain complexity rather than program complexity. There are some possible situations where the CDC metric satisfies property 7, and others where it does not. For example, if the CDC metric is calculated using information derived only from the comments of a class, then it meets property 7 if the comments are multi-line comments, where some comment sentences stretch over multiple lines. It does not meet property 7 if all the comments are single line comments. In a similar situation, properties 6 and 9 are satisfied if, when the programs are concatenated, the comments documenting the programs are updated to address the interactions between the programs. The CDC metric does not meet property 8. The CDC metric measures domain complexity, or psychological complexity, while property 8 specifically avoids measuring psychological complexity. We note here, as well, that the CDC metric can be used as part of a quantification of the usefulness of mnemonics, which Weyucker specifically says property 8 does not address.

The Semantic Class Definition Entropy metric is described in detail elsewhere [12]. This metric has been partly validated, with good initial results [12].

4.3 Relative class complexity and key class identification

Most often, what is more important to a software designer than a general complexity measure for a class, is a relative comparison of the complexity of a particular class to all other classes in the same system. For this reason, we define Relative Class Domain Complexity:

RCDC = CDC/ maximum CDC for any class in the same system

Where CDC = Class Domain Complexity, as defined above

Therefore, the class with maximum domain complexity in the entire system will have an RCDC of 1, and classes with similarly large domain complexity will have values near 1. Uncomplex classes will have low RCDC, nearer 0.

This can be used to detect *key* classes in the system. We define a metric for key class identity, based on the RCDC metric:

KCI = { (0 or 1) | (0 when RCDC < 0.75, 1 when RCDC >= 0.75) }

The cutoff for a key class, 0.75, is an initial rule of thumb, and should be verified empirically. The analysis relative to Weyucker's properties for RCDC and for KCI is the same as for the CDC metric.

4.4 Class interface complexity

We define two different metrics for class interface complexity, which we call CIC and SCIDE. CIC and SCIDE measure the complexity of the interface methods of a class (in C++, the public functions of the class). CIC is defined similarly to CDC, but over the interface methods of the class. Also, the Semantic Class Definition Entropy metric can be easily modified to measure interface complexity instead of class complexity; we call this metric Semantic Class Interface Definition Entropy (SCIDE).

4.5 Class overlap

We define two class overlap measures, Class Overlap A (COa), and Class Overlap B (COb) that determine the overlap of functionality between two classes by examining the concepts that appear in more than one class. For example, if concept A appears in both class 1's set of conceptual graphs and the set of conceptual graphs belonging to class 2, then classes 1 and 2

overlap on concept A. In our current definition, a concept A in class 1 is considered to be the same concept as a concept B in class 2 only if either:

Option 1 concept A is exactly the same as concept B (same concept, in same location), **or**

Option 2

- Type (Concept A) = Type (Concept B), **and**
- If referent (Concept A) is the individual marker i, then referent (concept B) is either i or *, the generic concept.

Thus,

COa = (1/M) $\Sigma_{j=1}^{N-1}$ ($\Sigma_{i=j+1}^{N}$ **| overlap of pairs of classes c_i and c_j | / |total number of possible overlaps|**)

Where **|overlap of pairs of classes c_i and c_j |** = number of concepts that belong to the conceptual graphs of both c_i and c_j.

|total number of possible overlaps| = minimum number of concepts (c_i, c_j) (if $|c_i| \leq |c_j|$, then total number of possible overlaps = $|c_i|$, else total number of possible overlaps = $|c_j|$).

N = number of classes in the system

M = number of pairs of classes = N choose 2 = N!/2! (N-2)! = N (N-1)/2

An alternate definition of COa is possible, if the first part of Option 2 in the concept overlap definition is modified to read: Type(Concept A) $\cap$ Type (Concept B). Whether this is a useful modification of COa is a topic for further study.

Although this metric measures overlap of class functionality, it has much in common with cohesion metrics, and thus is evaluated with respect to the properties of cohesion metrics. This metric meets all four cohesion properties of Briand et al. [5].

This metric has a drawback in that it considers only concepts and not conceptual relations. If both are considered, then the problem becomes one of detecting overlap by performing the intersection of conceptual graphs associated with each method. This leads to the definition of overlap metric COb, which has a formulation similar to that of LORM3, but which is not presented here due to space considerations.

4.6 Documentation

The documentation quality of a class can be measured by examining the level of useful information provided by the identifiers of the class (names of methods, attributes, etc.), and by examining the level of useful information provided by the comments in the class. Before now, the only determiner for comment quality was a count of comments, and there was no automated determination possible for identifier quality. However, many software developers think that a class is best documented by the use of good identifier names [6, 26]. The use of semantic metrics can provide both. We define the following metrics, Overall Class Documentation Quality A (OCDQa), Overall Class Documentation Quality B (OCDQb), the Class Comment Quality (CCQ), and the Class Identifier Quality (CIQ), not presented here due to space limitations.

5. Conclusions

A new object-oriented semantic metrics suite has been presented . The metrics of this suite provide a higher level, semantic, domain oriented view of object-oriented software than is possible employing traditional, syntactically-oriented object-oriented metrics, and thus can be more accurate in many cases than syntactic metrics. Some quantities, such as key class identification, class overlap, and documentation quality, can be measured with these metrics,

whereas measuring these quantities with syntactic metrics was difficult or impossible.

The semantic metrics presented here have been evaluated by the use of Weyucker's properties [26] and Briand et al.'s cohesion properties [5], as appropriate. Validation of these metrics is underway. Early validation efforts have been good [12].

REFERENCES

[1] Albrecht, A.J., and G.E.Gaffney. Software Function, Source Lines of Code and Development Effort Prediction: A Software Science Validation. *IEEE Trans. on Software Engineering*, Nov. 1983, pp. 639-648.

[2] Bansiya, Jagdish, Davis, Carl, and Etzkorn, Letha,.An Entropy Based Complexity Measure for Object-Oriented Designs. *Theory and Practice of Object Systems*, 5, 2, May, 1999, pp.1-9.

[3] Bansiya, J., Etzkorn, L., Davis, C., and Li, W. A Class Cohesion Metric for Object-Oriented Design. *Journal of Object-Oriented Programming*, 11, 8, Jan.,1999, pp. 47-52.

[4] Basili, V., L. Briand, and W.L. Melo,. A validation of object-oriented metrics as quality indicators. *IEEE Trans. on Software Engineering*, 22, 10, Oct. 1996, pp.751-761.

[5] Briand, L.C., Daly, J.W.,and Wust, J. A Unified Framework for Cohesion Measurement in Object-Oriented Systems. *Empirical Software Engineering*, 3, 1, 1998, pp. 65-115.

[6] Chidamber, S.R., and Kemerer, C.F. A Metrics Suite for Object-Oriented Design. *IEEE Trans. on Software Engineering*, 20,6, 476-493,1994.

[7] Chidamber, S.R., and Kemerer, C.F. Toward a Metrics Suite for Object-Oriented Design. *Proc. of OOPSLA*, 1991, pp. 197-211.

[8] Davis, J., & LeBlanc, R. A Study of the Applicability of Complexity Measures. *IEEE Trans. on Software Engineering*, 14, Sept, 1988, pp. 1366-1372.

[9] Eder, J., Kappel, G., and Schrefl, M.. Coupling and Cohesion in Object-Oriented Systems. *Technical Report*, University of Klagenfurt, 1994.

[10] Etzkorn, Letha, Bansiya, Jagdish, and Davis, Carl. Design and Code Complexity Metrics for OO Classes. *Journal of Object-Oriented Programming*, 12,1, March,1999, pp. 35-40.

[11] Etzkorn, Letha, Bowen, Lisa, and Davis, Carl. An Approach to Program Understanding by Natural Language Understanding. *Natural Language Engineering*, accepted, to be published in 5, 1, 1999, pp.1-18.

[12] Etzkorn, L.H., Bansiya, J., and Davis, C.G.. A Semantic Entropy Metric. currently in final development.

[13] Etzkorn, L.H., and Davis, C.G. Automated Object-Oriented Reusable Component Identification. *Knowledge-Based Systems*, 9, 8, Dec. 1996, pp. 517-524.

[14] Etzkorn, L.H., and Davis, C.G. Automatically Identifying Reusable OO Legacy Code. *IEEE Computer*, 30,10, 1997, pp. 66-71.

[15] Etzkorn, L.H., and Davis, C.G. A Documentation-related Approach to Object-Oriented Program Understanding. *Proc. of the IEEE Third Workshop on Program Comprehension*, IEEE Computer Society Press, Los Alamitos, CA, 1994, pp. 39-45.

[16] Etzkorn, L.H., Davis, C.G., Bowen, L.L., Etzkorn, D.B., Lewis, L.W., Vinz, B.L., and Wolf, J.C.. A Knowledge-Based Approach to Object-Oriented Legacy Code Reuse. *Proc. of the Second IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '96)*, IEEE Computer Society Press, Los Alamitos, CA, 1996, pp. 39-45.

[17] Etzkorn, L., Davis, C., and Li, W. A Practical Look at the Lack of Cohesion in Methods Metric. *Journal of Object-Oriented Programming*, 11, 5, Sept. 1998, pp.27-34

[18] Harrison, W. An entropy-based measure of software complexity. *IEEE Trans. on Software Engineering*, 18, Nov., 1992, pp.1025-1029.

[19] Henderson-Sellers, Brian. *Object-Oriented Metrics: Measures of Complexity*, Prentice-Hall, Upper Saddle River, NJ, 1996.

[20] Hitz, M., and Montazeri, B. Chidamber and Kemerer's Metrics Suite: A Measurement Theory Perspective. *IEEE Trans. on Software Engineering*, 22, 4, April, 1996, pp. 267-271.

[21] Li, W. Another Metric Suite for Object-Oriented Programming. *Journal of Systems and Software*, 44, 1998, pp. 155-162.

[22] Li, W., Etzkorn, L., Davis, C., and Talburt, J. An Empirical Study of Design Evolution in a Software System. *Information and Software Technology*, accepted, to appear 2000.

[23] Li, W., and Henry, S. Object-oriented Metrics that Predict Maintainability. *The Journal of Systems and Software*, 23, 2, 111-122 (1993).

[24] Lorenz, M., and Kidd, J. *Object-Oriented Software Metrics*, PTR Prentice-Hall, Englewood Cliffs, NJ, 1994.

[25] Sowa, J.F. *Conceptual Structures: Information Processing in Mind and Machine*, Addison-Wesley, 1984.

[26] Weyucker, E.J. Evaluating Software Complexity Measures. *IEEE Trans. on Software Engineering*, Vol. 14, No. 9, Sept. 1988, pp. 1357-1365.