Çankaya Üniversitesi Fen-Edebiyat Fakültesi,
Journal of Arts and Sciences Sayı: 5, Mayıs 2006

Complexity Metrics AS Predictors of Maintainability and Integrability of Software components

*Nael SALMAN**

Abstract

The work presented in this paper introduces a set of metrics for component oriented software systems. The work focuses mainly on the complexity that results mainly from factors related to system structure and connectivity. Also, a new set of properties that a component-oriented complexity metric must possess are defined. The metrics have been evaluated using the properties defined in this paper. A case study has been conducted to detect the power of complexity metrics in predicting integration and maintenance efforts. The results of the study revealed that component oriented complexity metrics can be of great value in predicting both integration and maintenance efforts.

1. INTRODUCTION

Use of Abstraction, as the key to the identification of system building blocks or components has been one of the most significant and critical issues to attract interest from programming language designers, methodologists, and software developers since the early days of software development. The earliest work started with process abstraction, which was not powerful enough to support the building of large and complex programs. Then appeared the data processing view, emphasizing function abstraction that receives inputs when called, does processing in its body and yields a value as output (Watt, Findlay, and Hughes, 1990), (Sebesta, 2002). Later, and particularly during the 1980's, the object-oriented (OO) approach arrived and introduced a different unit of abstraction which encapsulates both data and functions. The fundamental building block in the OO approach is "the class" which

* Çankaya University, Department of Computer Engineering, Ankara, email: nsalman@cankaya.edu.tr

is a collection of objects, and has the power to hide information from its clients. The class abstraction allows the building of large and complex systems as hierarchies of objects (Sebesta, 2002). Most recently, the component oriented (CO) system development approach emerged with software components as its principal building block.

Component-Oriented Software Engineering (COSE) is likely to become the main and preferred stream for software development (Ravichandran, and Rothenberer, 2003). The CO paradigm focuses on developing large software systems by integrating prefabricated software components (Dogru and Tanik, 2003). It facilitates the process of software development (Vitharana, Zahedi, Jain, 2003) and solves many adaptation and maintenance problems (Basili and Boehm, 2001). It is quite clear that in the last few years research has focused on methods and approaches that work towards developing software systems by integrating already developed components.

In contrast, very little effort has been devoted for metrics and measures that can be used to evaluate design complexities, integration complexities, and overall system complexities for systems that are developed using COSE methods. Also, theoretical foundations for measuring and validating measurements have not emerged yet. The customizability and reliability of prefabricated components is a critical issue, requiringnew metrics to measure these attributes. Cost estimation, productivity estimation, reliability, and maintainability measurements are still open research fields for COSE (Vitharana, Zahedi, and Jain, 2003). The main issues in COTS metrics are capturing integration complexity; and complex interfaces tend to complicate the testing process of the system (Sedigh-Ali, Ghafoor, and Paul, 2001). Measuring the degree of structuredness in software systems is an important issue since system organization will necessarily have an impact on maintainability (Visaggio, 1997). Productivity is still significantly affected by personnel skills in COTS systems so guidelines should be introduced to help developers in increasing productivity in COTS based systems (Basili and Boehm, 2001).

The research reported here introduces a set of metrics that is intended to measure the new features related to the introduction of the CO approach. We only focus on metrics that can be collected during early stages of system development, particularly from system design documents. A metrics validation approach for component oriented systems is also introduced.

The rest of the paper is organized as follows: In section 2 preliminary definitions of the key concepts that are used in this article are introduced, in section 3 CO structural complexity and a set of metrics characterizing it are defined. In section 4

a set of properties that a CO complexity metric must possess are described. In section 5 metrics evaluation validation approach is performed. In section 6 we derive our conclusions, directions for future work and probable extensions to this paper.

2. PRELIMINARY DEFINITIONS

Before presenting component-oriented software complexity or introducing our metrics validation approach, we will introduce our perception of the concepts related to the subject. Our aim in introducing these definitions is to avoid any confusion that may arise.

Component Oriented Software System: this is a software system that is modeled and designed to be developed by integrating components of independent deployment. It is also necessary to point out that the system is totally modeled using component oriented software engineering modeling language (e.g. COSEML which is a dedicated modeling language for component oriented software modeling (Dogru and Tanik, 2003)). In order to familiarize the reader with this and to clarify any possible ambiguity, we present a COSEML model for a simplified university information system in Figure 1.

Software Component: Many definitions of this term can be found in different sources. The definition that combines most of the important features that appear in other definitions and seems to be widely accepted is the one given in (Szyperski, Gruntz, and Murer, 2002). They define a software component as:

Coupling Between Components: Two components are coupled if there is a link between them, where a link means a request for a service. The direction of the link indicates which component requests the service or is dependent on the other. Hence, coupling is considered as a measure of inter-component dependency. It is widely accepted that excessive coupling is not a good design practice and usually results in complex systems that are difficult to maintain and upgrade.

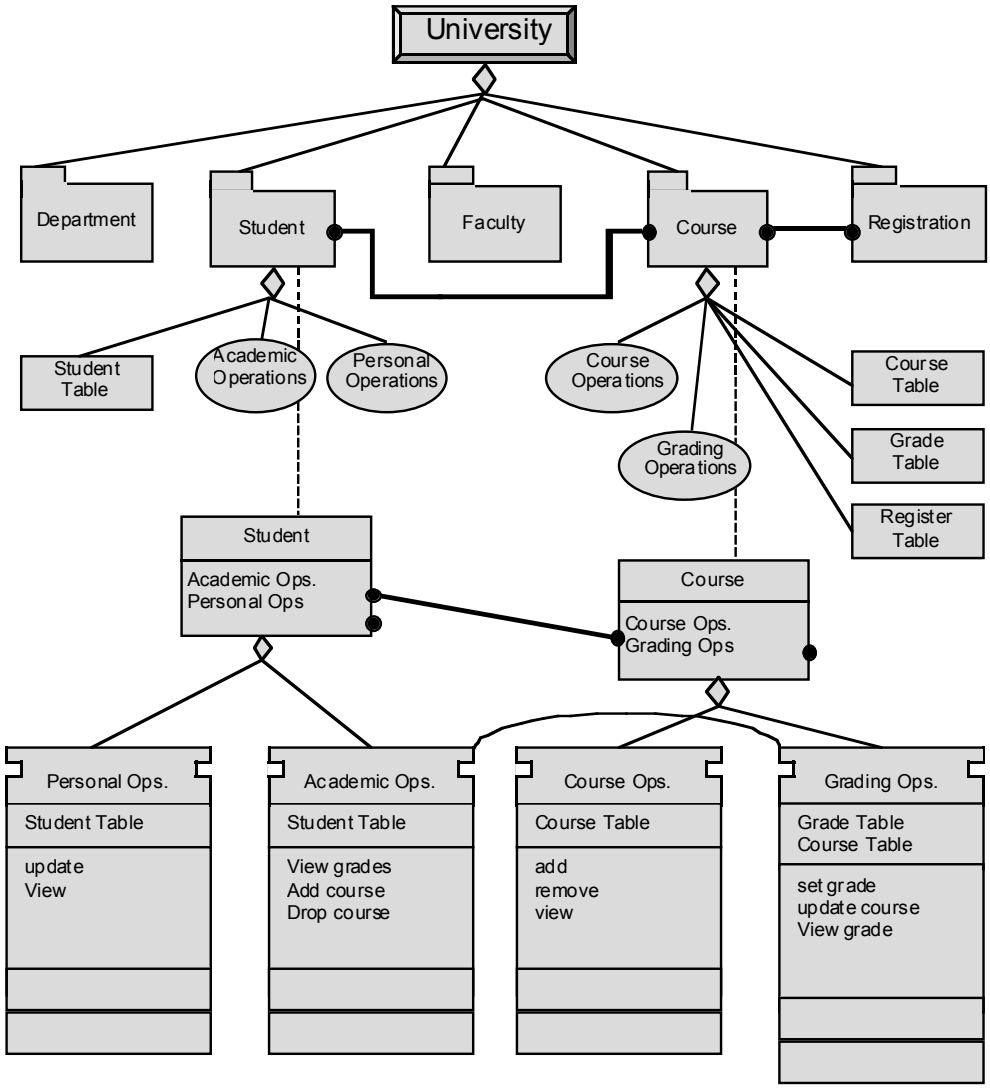

Figure 1. Simplified University System Modeled Using COSEML

CO System Structure: Software system structure is defined as the way through which system building elements are organized in relation to each other and their environment (Gorla and Ramakrishnan, 1997). It deals with methods that can be applied to achieve maximized reusability and reliability (Clements, 1995), (Clements et al., 1995). From these two views of system structure we can conclude that it is a design decision. Two or more different design alternatives may result in

different structures. Intuitively, different structures of the same system will certainly lead to different values of structural complexity.

Structural Complexity: This term has been defined and interpreted in many different ways. While some view complexity as a factor mostly related to size and functionality, others relate complexity to the degree of mental effort required to understand (Zuse, 1993). Tian and Zelkowitz (1995) considered software complexity as the aspect of software that is used to predict external properties of the program (e.g. reliability, understandability, maintainability). Since the main focus of COSE is on structure (Dogru and Tanik, 2003), we are dedicating more interest to structural complexity of systems.

Component Composition: The process of integrating two or more components with well-defined interfaces to produce a single functional component.

3. COMPLEXITY METRICS

A component-oriented software system can be obtained as a result of the composition of some components with defined interfaces (Szyperski et al., 2002). A component's functionality is implemented in its methods and is provided for other components through its well-defined interfaces. Based on this view and the preliminary definitions presented in section 2, component-oriented software systems structural complexity can be characterized using the following attributes.

1- components in the system
2- connectors between components
3- interfaces of each component
4- composition tree

A set of metrics that characterize these attributes are defined as follows:

1- To characterize the components in a system the following metrics are defined:
   a. Total number of components (TNC) in a system is defined as: the count of all components in the system that appear in different levels of abstractions.
   b. Average number of methods per component (ANMC) metric is used. This metrics is estimated by dividing the total number of methods by the total number of components.
   c. Total number if implemented components (TNIC): The count of implemented components only.

2- To characterize connectors in a system, the following metrics are defined
   a. Total number of links (TNL): count of all links appearing in the system design model in all levels.
   b. Average number of links between components (ANLC): Total number of links divided by the total number of components.
   c. Average number of links per interface (ANLI): Total number of links between interfaces divided by the total number of interfaces.
3- To characterize interfaces in the system, the following metrics are defined:
   a. Total number of interfaces (TNI): count of all interfaces of all components in the system.
   b. Average number of interfaces per component (ANIC): Total number of interfaces divided by the total number of components.
4- To characterize the composition tree the following metrics are defined:
   a. Depth of the composition tree (DCT): count of the number of levels of the composition tree.
   b. Width of the composition tree (WCT): count of the maximum number of components in any level of the composition tree.

4. PROPERTIES OF A SOFTWARE COMPLEXITY METRIC

A software complexity metric is valid if it succeeds in satisfying defined properties. Several researchers have tried to describe a set of properties that a good software complexity metric must satisfy. Such approaches can be seen in the works described in (Weyuker, 1988), (Briand, Morasca, and Basili, 1996), (Kitchenham, Pfleeger, and Fenton, 1995), (Tian, and Zelkowitz, 1995), (Schneidewind, 1992) and (Zuse, 1996), but valid approaches are not limited to these. While some of the work has gained more popularity than the others (Weyuker, 1988), none has been totally accepted or totally rejected by the software development community. Since no global acceptance has been reached for any of these properties and, also, none of these properties have specifically tackled the particular and new aspects of component-oriented software systems, we introduce a set of properties that a component-oriented system complexity metric must satisfy. The properties defined in this article came as a result of investigating the properties described in (Weyuker, 1988), (Briand, Morasca, and Basili, 1996), (Kitchenham, Pfleeger, and Fenton, 1995), (Tian, and Zelkowitz, 1995), (Schneidewind, 1992), and (Zuse, 1996). The properties described in this paper do not have a generic nature in the sense that we

do not claim that they can apply to all types of complexity metrics ,especially those proposed for "non-component-oriented" systems.

Property 1: *Nonnegativity*: A complexity metric value can not be a negative number. For some complexity metrics it is necessary to be even stricter, since a value of zero will not always be accepted.

Interpretation guidelines: The meaning of a complexity metric value for a software artifact (a software artifact can be a method, component, or the whole system) that provides some functionality to be equal to zero is that the artifact is the least-complex possible design that can provide that functionality. A lower complexity value, for two functionally equal designs, is preferred over a higher value since lower complexity is believed to be associated with less development, testing, and maintenance efforts.

Property 2: *Scalability*: A software complexity metric must provide a scale of values. Comparison between different alternatives must be possible. For any two software artifacts it must be possible to compare and then make managerial decisions according to the metrics values. For any two functionally-equal components C_1 and C_2, if Complexity(C_1) > Complexity(C_2) then C_2 is preferred over C_1 assuming that we keep all other parameters constant. This is due to the fact that C_2 will require less testing, less integration, and less maintenance efforts. Also, metrics must provide enough information to help managers make business decisions and compare different alternatives.

Property 3: The complexity of a single software unit *S* composed of two software components can not be less than the sum of the complexities of the individual components.

$$\text{Complexity}(S) >= \text{Complexity}(C_1) + \text{Complexity}(C_2)$$

According to the metrics described in section 4, the complexity of a component-oriented software system is a function of the complexities of individual components that make it up, and an added complexity will appear as a result of new interactions that may exist between the components. In the best case, when a system is composed of two components and no new added interactions between the components are available, the system's complexity will be equal to the sum of the individual component complexities.

Property 4: If a component C is decomposed into two or more components C1, C2, .., Cn then the sum of complexities of the resulting components is no more than the overall complexity of the original component.

$$\text{Complexity}(C_1) + \text{Complexity}(C_2) + \ldots + \text{Complexity}(C_n) <= \text{Complexity}(C)$$

The reason for this is that, according to our perception of the three-level component-oriented software complexity, there is usually an added complexity whenever two components are composed. The new complexity usually results from the interactions between these components. So, when the component is decomposed these links will disappear and only the component's intrinsic complexity will remain.

Property 5: The complexity value of one component does not have a direct relation to its functionality, i.e. for any two components C_1 and C_2, if Complexity(C_1) > Complexity(C_2) then it is not necessary that C_1 provides more functionality than C_2. The same functionality can be obtained by different designs and then implementation. The complexity measures described in this article are those that enable software developers and/or managers to take decisions and contrast/compare different alternative solutions to the same problem. Of course, any added functionality may introduce an added complexity. So, a complexity metric does not consider evaluating functionality of the system or provide any information about the system size.

Property 6: The complexity value is directly influenced by structure. Two different structures for the same functionality can result in two different complexity values. A complexity measure of the system can have different values for different alternative architectures of the same functionality.

5. METRICS EVALUATION AND VALIDATION

The metrics have been evaluated using the set of properties defined in section 4. All metrics satisfied all properties. This leads to concluding that metrics qualify from a mathematical perspective. What remains is examining whether metrics qualify form a practical perspective. To examine the latter we conducted a case study where data from 25 graduate students' projects have been collected. All of the projects are designed for component oriented software development using COSEML (Dogru and Tanik).

The case study has been conducted to investigate the potential of using complexity metrics values as predictors of both integrability and maintainability of component oriented systems. Integrability is defined as the total effort spent on defining inter-component link and component interfaces. Maintainability is defined as the effort spent on making corrections to errors discovered during the design phase of system development.

Data Collection approach: Data were collected from the design documents. A

metrics collection form was prepared for the purpose of making the metrics collection less costly and to encourage the developers to perform the task. The developers were exposed to a short training course on how to collect these metrics from the design documents.

Regression analyses have been performed where all metrics values were fed in a forward addition manner. The results obtained from the regression analyses can be summarized as follows:

1) Correction Effort per Component (Component Maintainability) regression analysis: Variables have been fed to the model were fed in a forward addition manner. Variables whose coefficients' corresponding p-values are less than or equal to 0.05 are added to the model. The regression analysis produced the following regression model:

Correction-Effort Per Comp = exp(-0.02 * (TNC) + 0.07* (TNIC)+ 0.2 * (ANMC) – 2.37)

(p = 0.003) (p = 0.008) (p = 0.04) (p = 0.0)

The model demonstrates a high statistical significance with maximum p-value 0f 0.04 and R2 0f 0.95. Besides being statistically significant, the model also is practically significant as well.

The average error rate is encouraging to recommend the model for practical use with a value of 13% when removing the outliers. The plot of the regression model is shown in figure 2.

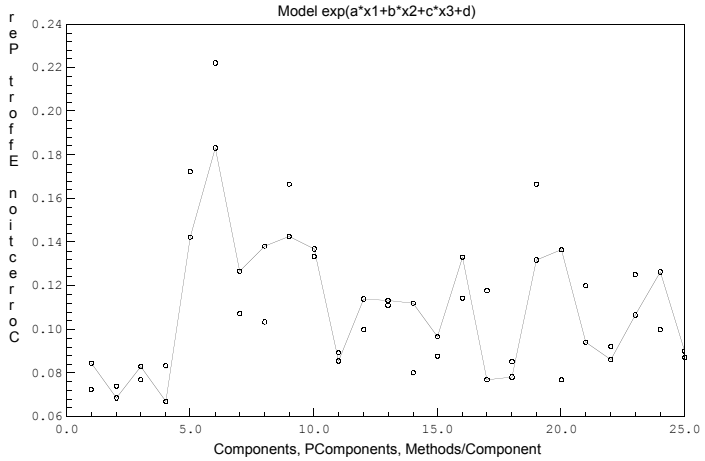

Figure 2: Correction Effort Per Component Regression Model Plot

2) Integration effort (Integrability) regression analysis: Variables have been fed to the model were fed in a forward addition manner. Variables whose coefficients' corresponding p-values are less than or equal to 0.05 are added to the model. The regression analysis produced the following regression model:

Integration Effort = 0.1 * (TNL) + 2.6
(p = 0.0) *(p = 0.0)*

The model uses only the total number of links measure. We believe that some other variables must be related to integration effort e.g. number of interfaces and number of components which are intuitively believed to influence integrability.

Despite the fact that the model contains only one variable, it still bears both statistical and practical significance. Both p-value of the variable coefficient and the constant are equal to 0.0. R2 has a value of 0.91 which is also quite high to encourage the adoption of the model. Average error rate is 11% when removing outliers. The model plot is shown in figure 3.

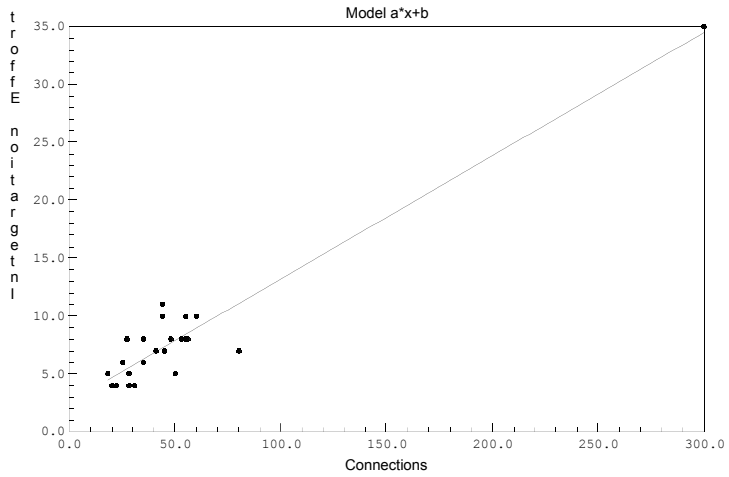

Figure 3: Integration Effort Model Plot

6. CONCLUSIONS AND FUTURE WORK

Structural complexity of component oriented systems has been defined. Metrics for measuring CO systems' structural complexity have been also been defined and validated. The results obtained are of great significance and worth consideration for further research in the field. Making early decisions about integrability and maintainability during system design stage has been a matter of great interest to researchers since the early days of software development.

The study still needs further validation using data from industry practices. Using implemented projects will provide opportunities to examine the relationships between metrics values and several other product quality factors like performance and reliability.

There is still little effort dedicated to component oriented software engineering measurement in both the theoretical and empirical grounds. The major problem that researchers encounter in the field is a lack of experimental data from the industry that can be used to validate the proposed measures and/or measurement frameworks.

References

Basili, V. R., and Boehm, B. 2001. COTS-Based Systems Top 10 List. IEEE Computer, vol. 34, No. 5: 91—93.

Briand, L. C., Morasca, S., and Basili, V. R. 1996. Property-Based Software Engineering Measurement. IEEE Transactions on Software Engineering, vol. 22, No. 1: 68—86.

Clements, P. C., 1995, "From Subroutines to Subsystems: Component-Based Software Development," American Programmer, vol. 8, no. 11.

Clements, P. C., Bass, L., Kazman, R., Abowd, G., 1995, "Predicting Software Quality by Architecture-Level Evaluation," *Proceedings of the Fifth International Conference of Software Quality*, Austinx, Tx, October, 1995.

Dogru, A. H., and Tanik, M. 2003. A process Model for Component Oriented Software Engineering. IEEE Software, March/April: 34—41.

Gorla, N., Ramakrishnan. R., 1997, Effect of Software Structure Attributes on Software Development Productivity, *Journal of Systems and Software*, vol. 36:191-199

Halstead, M. 1997. Elements of Software Science. Elsevier Computer Science Library.

Kitchenham, B., Pfleeger, S. L., and Fenton, N. 1995. Towards a Framework for Software Measurement Validation. IEEE Transactions on Software Engineering, vol. 21, No. 12: 929—944.

Kitchenham, B., Pfleeger, S. L., Pickard, L. M., Jones, P. W., Hoaglin, D. C., El Emam, K., and Rosenberg, J. 2002. Preliminary Guidelines for Empirical Research in Software Engineering. IEEE Transactions on Software Engineering, vol. 28, No. 8: 721—734.

Mendonça, M. G., and Basili, V. R. 2000. Validating of an Approach for Improving Existing Measurement Frameworks. IEEE Trans. on Software Engineering, vol. 26, No. 6: 484—499.

Ravichandran, T., and Rothenberger, M. 2003. Software Reuse Strategies and Component Markets. Communications of the ACM, vol. 46, No. 8: 109—114.

Schneidewind, N. F. 1992. Methodology for Validating Software Metrics. IEEE Transactions on Software Engineering, vol. 18, No. 5: 410—422.

Sebesta, R. W. 2002. Concepts of Programming Languages. 5th Ed., Addison Wesley.

Sedigh-Ali, S., Ghafoor, A., and Paul, R. A. 2001. Software Engineering Metrics for COTS-Based Systems. IEEE Computer, vol. 34, No. 6: 44—50.

Szyperski, C., Gruntz, D., and Murer, S. 2002. Component Software - Beyond Object-Oriented Programming. 2nd Ed. Addison-Wesley / ACM Press: 1—47.

Tian, J., and Zelkowitz, M. V. 1995. Complexity Measure Evaluation and Selection. IEEE Transactions on Software Engineering, vol. 21, No. 8: 641—650.

Vitharana, P., Zahedi, F. M., and Jain, H. 2003, "Design Retrieval and Assembly in Component-Based Software Development. Communications of the ACM, vol. 46, No. 11: 97—102.

Visaggio, G. 1997. Structural Information as a Quality Metric in Software Systems Organization. Proceeding of ICSM: 92—99.

Watt, D. A., Findlay, W., and Hughes, J., 1990. Programming Language Concepts and Paradigms," Prentice Hall.

Weyuker, E. J., 1988. Evaluating Software Complexity Measures. IEEE Transactions on Software Engineering, vol. 14, No. 9: 1357—1365.

Zelkowitz, M. V., and Wallace, D., 1997. Experimental Validation on Software Engineering. Information and Software Technology, Elsevier Sciences, vol. 39: 735—743.

Zelkowitz, M. V., and Wallace, D. 1998. Experimental Models for Validating Technology. IEEE Computer, Vol.31, No.5: 23—31.

Zuse, H. 1993. Criteria for Program Comprehension Derived from Software Complexity Metrics. Proceedings of the Second International Workshop on Software Comprehension, IEEE, Capri/Italy: 8—16.

Zuse, H. 1996. Foundations of Object-Oriented Software Measures, IEEE Proceedings of METRICS'96: 75—88.