COMPLEXITY MEASURE IN COMPONENTS OF SOFTWARE

Uma Singh*
Bhawna Malik**

ABSTRACT

The complexity of a software or a dynamical process expresses the degree to which components engage in organized structured interactions. High complexity is achieved in systems that exhibit a mixture of order and disorder (randomness and regularity). Each component introduces properties such as constraints associated with its use, interactions with other components and customizability properties. The ability to measure system dependency implies the capability to locate weakness in the system designand to determine the level of software quality. Measures of complexity allow different systems to be compared to each other by applying a common metric. This paper proposes an interface complexity metric for software components, which is based on complexity involved in the interface methods and properties used in the interface.

Index Terms: *Software Complexity, Software metric, Information flow.*

INTRODUCTION

Software complexity is defined as follows:

"A software complexity is a term that encompasses numerous properties of a piece of software, all of which affect internal interactions."

The term software complexity is often applied to the interaction between a program and a programmer working on some programming task. Each component in a CBS's Structure contributes a specific function towards the services provided by the systems. As soon as a new component evolves, it brings a change in the composite functionality of system. Benefits of Component-based development include:

• Lower cost of development and shorter delivery schedules
• Better reliability and reduced maintenance costs
• Lets developers focus on their business

Requirements and core competencies, rather than re-solving the same technical problems over and over.

• Provides extensibility because components can be assembled into many different configurations to provide unique variants of a system as needed.

(This is especially common today for industries such as cellular technology, consumer electronics, and automotive systems).

• Components that use different languages and technologies can be mixed and matched.

• Higher level models make complex systems easier To understand component based development is the Best technique for managing complexity of systems as they increase in size and scope. Dependency conflicts arise due to continuous update and modification of current systems.Even after the execution of special uninstall procedures to remove the problem application, junk libraries and files might remain in the system. The main cause of these difficulties is lack of a representation model for dependencies between the system components and application components along with mechanisms for managing these dependencies.Analysis of CBSS dependencies is an important part ofsoftware researchfor understandability, testability, maintainability and reusability of a component based system. Thus, dependency metrics could have a realimpact on the quality of the system delivered to the user.

METRICS OF COMPONENT BASED SYSTEMS OVER TRADITIONAL SOFTWARE METRICS

Traditional metrics has to be redefined or enhanced tocomply with component-based software development. Traditional metrics have been appliedto the measurement of software complexity of structured systems since 1976. Some commontraditional software metrics are:

- source lines of code (SLOC)
- cyclomatic complexity
- functionpoint analysis (FPA)
- bugs per lines of code and
- code coverage.

Traditional software metrics are usually applicable to small programs, whereas the metrics for CBSS should depend mainly on the granularity and interoperability aspects of the components. Size of a component is normally not known to the component developers, whereas most of the traditional size metrics such as SLOC and bugs (faults)/code line and code coverage are based on lines of code, which is not applicable to CBSS.

Traditional cyclomatic complexity metric suite cannot be applicable either in CBSS becauseoperator and operand counts are not known in CBSS and the number of linearly independent paths cannot be measured. FPA depends on the weights that were developed in a particular environment, which arises about the validity of this method for general application even though some improved measures like adjusting the counting method have been taken .There are many inherent differences in CBSS and non-CBSS so that the traditional software metrics are inappropriate for CBSS. Besides, the traditional software metrics do not address the interface complexities and integration-level metrics, which are also not applicable to CBSS.

Nature of software complexity

That complexity is strongly connected to the amount of resources needed in a project is something that is stated by most of the researchers of software metrics. The notion is that a more complex problem or solution is demanding more resources from the project, in form of man-hours, computer time, support software etc. A large share of the resources is used to find errors, debug, and retest; thus, an associated measure of complexity is the number of software errors. Based on the literature (Fenton & Pfleeger, 1996; Zuse, 1991; Ohlsson, 1996) we would like to suggest that software complexity is made up of the following parts:

1. **Problem Complexity** (Also called **computational complexity**) measures the complexity of the underlying problem. This type of complexity can be traced back to the requirements phase, when the problem is defined. If the problem can be described with algorithms and functions it is also possible to compare different problems with each other, and try to state which problem is the most complex.
2. **Algorithmic Complexity** reflects the complexity of the algorithm implemented to solve the problem. This is where the notion of efficiency is applied. By experimentally comparing different algorithms we can establish which algorithm gives the most efficient solution to the problem, and thus has the lowest degree of added complexity. This type of complexity is measurable as soon as an algorithm of a solution is created, usually during the design phase. However, historically algorithmic complexity has been measured on code, where the mathematical structure is more apparent.
3. **Structural complexity** measures the structure of the software used to implement the algorithm. Practitioners and researchers have recognized for a long time that there may be a link between the structure of the products and their quality. In other words, the structure of requirements, design, and code may help us to understand the difficulty we sometimes have in converting one product to another (as, for example, implementing a design as a code), in testing a product (as in testing code or validating requirements, for instance) or in predicting external software attributes from early product measures.

4. **Cognitive complexity** measures the effort required to understand the software. It has to do with the psychological perception or the relative difficulty of undertaking or completing a system. However, since this aspect of complexity is more an object of study for the social scientists and psychologists, we are not considering it in this report. Nonetheless, it is important to notice that the human mind is a constraint for the software process, and that it influences the attributes of the software we want to measure, such as quality and productivity.

Structural Complexity

Algorithmic complexity

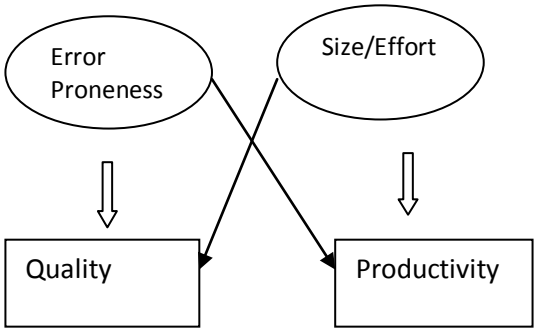

Fig.1 Simplified model of complexity

Complexity in Components

In literature, software complexity has been defined differently by many researchers. IEEE defines software complexity as "the degree to which a system or component has a design or implementation that is difficult to understand and verify". Complexity is defined in terms of execution time and storage required to perform the computation when computer acts as an interacting system. From time to time, many researchers have proposed various metrics for evaluating, predicting and controlling software complexity. Traditional software metrics have been designed and applied to the measurement of software complexity of structured systems since 1976. Halstead's software science metrics, Source line of code, McCabe's cyclomatic number and Kafura's& Henry's fan-in, fan-out, function point analysis, bugs or faults per line of code, code coverage are the best known early reported complexity metrics for traditional function-oriented approach. Bill Curtis has reported two types of software complexity – Psychological and Algorithmic. Psychological complexity affects the performance of programmers trying to comprehend or modify a class/module whereas algorithmic or computational complexity characterizes the run-time performance of an algorithm.Software complexity cannot be computed by a single parameter ofa component/program/software because it is multidimensional attribute of software. The prominent factors which contribute to complexity of a component-based software system are:

Size of each component: Size is also considered one of the parameter of program/class/component complexity. A class with more methods is harder to understand than a class with less number of methods and hence contributes more complexity.Large programs/components incur problem just by virtue of volume of information that must be absorbed to understand the program and more resources have to be used in their maintenance. So, size is a factor which adds complexity to a component.

Interfaces of each component: In CBSD, a component is linked with other components and hence has interfaces with them. Two or more components are said to be interfaced if there is a link between them, where a link means that a component submits an event and other components receive it. The direction of the link indicates that which component requests the services or dependent on the other. Interface between two components can be through incoming and outgoing interactions. These both types of interactions add complexity to a component-based software system.

Interfaces are the access points of component, through which a component can request a service declared in an interface of the service providing component. Mathematically, $I(Component_x)$ is defined as sum of complexity of the interface methods of the class. The complexity of interface methods depends on its nature. The nature of the interface methods are determined on the basis of their arguments and return types. Arguments and return types can have any of the three data types discussed earlier (primitive, user defined and structured). The weight values can be assigned to these methods by considering the total number of methods in each category. The different category methods have different value of weight. Weight of the method also depends on the number of methods in that category. If the number of methods are more then the weight value assigned will also increase. Now, Mathematically the Interface Factor, $I(Component_x)$, can be written as:

$I(Component_x) =$
$\sum_{i=1}^{m1} w_{simple_i} + \sum_{j=1}^{m1} w_{medium_j} + \sum_{k=1}^{m1} w_{complex_k}$
Where m1, m2 and m3 are the total number of interface
methods of simple, medium and complex nature respectively
By taking only interface complexity into account, an interface complexity measure for a component-based system is suggested as :
Average Incoming Interactions Complexity (AIIC) =

$$\frac{\sum_{i=1}^{m} II_i}{m}$$

Average Outgoing Interactions Complexity (AOIC) =

$$\frac{\sum_{i=1}^{m} OI_i}{m}$$

Average Interface Complexity of a Component Based System=
(AIC(CBS)) = $\frac{\sum_{i=1}^{m} II_i}{m} + \frac{\sum_{i=1}^{m} OI_i}{m}$
M = Number of components in the Component based
System (CBS)
II = Incoming Interactions
OI = Outing Interactions
i = Index variable

CONCLUSION AND DIRECTIONS FOR FUTURE RESEARCH

In this paper, first, we proposed a sets of metrics which characterize and evaluates the dependency between components, so that CBSS designers can identify critical components in terms of error-proneness and evaluate the impact of the change on the whole CBSS in terms of the difficulty of making a corrective change, which in turn allows designers to target components that need to be revised to improve the quality of the design. A linked list based approach is used to understand and manage the dependencies between components with the help of component dependency life cycle.Since it is difficult to calculate dependencies at all levels, an automated tool is required which can provide the number of dependencies at all levels.

REFERENCES

[1] C. Szyperski, Component Software: Beyond Object Oriented Programming, Second Editioned, Addison Wesley, New York, 2002,

[2] L. Narasimhan and B. Hendradjaya, "Some theoretical considerationsfor a suite of metrics for the integration of software components," Information Sciences, vol.177, 2007, pp. 844-64

[3] B. Li, " Managing dependencies in component-based systems based onmatrix model," Proc. Proceedings Of Net. Object. Days, Citeseer, 2003,pp.22-25

[4] A. Sharma, P.S. Grover and R. Kumar, "Dependency analysis forcomponent-based software systems," SIGSOFT Softw. Eng. Notes, vol.34, 2009, pp. 1-6

[5] Singh, R., Grover, P.S. (1997): A New Program Weighted Complexity Metric, Proc. International conference on Software Engg. (CONSEG'97), Chennai, India, pp. 33- 39.

[6] Harrison, W. (1982). Magel, K, Kluezny, R., dekock, A.: Applying Software Complexity Metrics to Program Maintenance, IEEE Computer, 15, pp. 65-79.

[7] A. De Lucia, A.R. Fasolino and M. Munro, "Understanding functionbehaviors through program slicing," wpc, 1996, pp. 9.

[8] S. Bates and S. Horwitz, " Incremental program testing using programdependence graphs," Proc. Proceedings of the 20th ACM SIGPLANSIGACT
symposium on Principles of programming languages, ACM,1993, pp.384-396

[9]Halstead, M.H. (1977): Elements of Software Science, New York: Elsevier North Holland.

[10] Chidamber, S. R., Kemerer, C.F. (1994): A Metrics Suite for Object Oriented Design, IEEE Transactions on Software Engineering, pp. 476-492.

[11]Kafura, D., Henry, S., 1981. Software Quality Metrics, Based on Interconnectivity,Journal of Systems and Software. Vol. 2, pp: 121-131.