

Measuring the Complexity of a UML Component Specification

Sajjad Mahmood

*Department of Computer Science
and Computer Engineering
La Trobe University
Melbourne, Australia
s3mahmood@students.latrobe.edu.au*

Richard Lai

*Department of Computer Science
and Computer Engineering
La Trobe University
Melbourne, Australia
lai@cs.latrobe.edu.au*

Abstract

Component Based System (CBS) development is about assembling individual components to produce a working system. However, its overall complexity does not only depend on the complexity of individual components. Further, component source code information is usually unavailable and they introduce additional properties such as constraints associated with its use, interactions among components, and customizability. The traditional complexity metrics are not adequate and do not easily apply to CBS as they mainly focus on either Lines of Codes (LOC) or information based on objects, classes and their inheritance properties. Recent CBS research suggests that most faults are found in few components. A complexity measure at specification level can be used for identifying these components; and precautionary actions can be taken to avoid the likely failures and to lower maintenance costs. There is therefore a need to develop a new technique for measuring the complexity of a component specification. This paper describes a complexity measure for a component specification written in Unified Modeling Language (UML).

1. Introduction

Software complexity is a key quality indicator that has been measured widely in software literature [1-5] and has proven impact on many software quality attributes such as maintainability, reliability and testability. Complexity assessment provides useful means to identify potential troublesome software components that require careful development and resource allocation. Since CBS emphasizes on assembling components in an interoperable manner, its complexity not only depends on individual

components but also on their interaction process. Traditional complexity metrics are not adequate for CBS because they are either dependent on LOC or measure complexity based on information about classes, objects and their inheritance properties. However, information on component's source code, class structure etc. is usually unavailable. Additional properties specific to a component are also introduced such as its interface structure, constraints associated with its usage, interactions among components; and its customizability and reusability properties. Thus, it limits the applicability of traditional metrics on CBS and needs a method that adequately considers these properties during a component complexity measure.

Recent research suggests that most faults are found in only a few of system's components [6]. If these components can be identified early at the specification level, then precautionary actions can be taken to avoid the likely failure causes and costly maintenance. Component complexity measure at the specification level provides an earlier quantitative assessments allowing more effective identification of fault prone components. A component specification consists of what a component provides and requires, (interfaces and precise definition of associated operations), the way in which the component is suppose to be interacted with (context dependencies and constraints), the possible roles that the component would play in a system, and the interaction between the components.

Interface plays a fundamental rule in defining component functionality by defining component's provided services. It acts as a primary source for component's understanding, use and implementation; and spells out the individual elements of a component in a syntactic manner. In addition to syntactic specification of a component, it is widely

acknowledged that semantic information about a component is necessary for its effective use. Examples of such information are the combination of parameters values an operation accepts and constraints on the order in which operations are invoked [7]. Besides component interface and constraints which define overall capability of a component, certain configurations are required for its proper usage. It involves looking at how a component plays different roles in a given context and how it may be used in different types of contexts [8]. Similarly, component interaction is a pivotal factor effecting overall complexity as it is a measure of the degree of interdependence between components.

To date, majority of CBS measurement research has been focused on identifying the key attributes of a CBS quality and the limitation of traditional measures when applied to CBS. Cho et al. [9] has proposed a suite of metrics based on complexity information of all classes and their methods which comprises each component and captures the dynamic complexity of a component based on analysis of component source code. However, one of the limitations of this technique is the absence of source code because of the black box nature of components. Narasimhan et al. [10] proposes component interaction complexity metrics by deriving packing density metrics and interaction density metrics. However, hardly any work has been done to derive software metrics from a component specification; in particular little work has been done to measure the complexity of a component specification based on a component's both internal and external properties.

This paper describes a complexity measure for a component specification written in UML [11] with a focus on identifying component attributes that affect its complexity at both component and intra-component levels. We identify that for a component specification, the overall complexity is attributed mainly to component interface, constraint and interaction. Our work is mainly motivated by the need to estimate component complexity based on quantitative metrics that can be evaluated early in the development phase. These metrics enables software analyst and developers to better understand the factors affecting complexity of a component and provides mechanism to identify complex components. It enables the complexity assessment at the specification level based on the analysis of both syntactic and semantic properties of a component.

2. Related Work

Several complexity measurement models and estimation techniques have been proposed in literature [2, 12-17]. Some of the conventional metrics such as LOC requires analysis of source code. The object oriented (OO) metrics focus on objects or classes and they measure complexity based on classes, methods and class hierarchy. Particular emphasis has been given to the measurement of design aspects to perform quality assessment early in the development process. However, these traditional metrics [2, 13, 18-21] are not adequate for CBD complexity measurement because of the unavailability of source code, difference of measurement unit and inadequate consideration of measurement factors.

Cho et al. proposes a suite of metrics for measuring the complexity, customizability and reusability of software component. Component complexity metrics is defined as a combination of four types of metrics, namely, component plain complexity, component static complexity, component dynamic complexity and component cyclomatic complexity [9]. These metrics need analysis of complexity for each class and method. It also needs analysis of source code of a component which is generally not available and it also makes it dependent on language of implementation.

Interface and middleware code required for integrating different components are two main factors effecting complexity of CBS [22]. Gill et al. [23] has extended this idea by identifying interface complexity metrics based on interface characterisation model of a component. They identify interface signature, interface constraints, interface packaging and configuration as factors contributing to overall interface complexity of a component. However, there is no discussion on how these identified attributes can be measured and validated.

Narasimhan et al. [10] have proposed a suite of metrics to measure complexity and criticality of components from the Component Interface Definition Language specification by deriving Component Packing Density metrics (CPD) and Component Interaction Density (CID). The CPD metrics relates component to the number of integrated components and CID metrics relates interactions between components to the number of available interactions in the entire system. One of the limits of this technique is that it only considers one individual attribute, component interaction, as a means of component

complexity measurement. Further, it only considers the frequency of interactions among components. However, in addition to interaction frequency, content of each interaction also has a significant effect on overall CBS complexity. It is important to consider the both the interaction frequency and complexity of information exchanged during an interaction.

3. Specification Complexity Measure

CBS departs from the conventional software development on it being integration centric as opposed to a development centric. In CBS, complexity not only depends on the individual components but also on underlying framework and integration process. To measure the complexity of an UML component specification, it is not practical to only consider one attribute of a CBS affecting its complexity. Therefore, to have reliable component complexity measure, it is necessary to identify and properly measure in detail each factor affecting its complexity.

One of the important objectives of UML component specification is to optimize the CBS quality. To provide a quantitative guide to the system analyst, our approach to complexity measure is based on component's internal properties and external behavior. This helps system analyst to identify error prone components and their interactions early in the specification phase of component based development life cycle. The complexity of a component specification can be attributed to a set of characteristics. After the detail analysis of component specification, we identify three factors: interface, constraints and interaction; as primary contributors to the complexity of a component.

3.1 Interface

Fundamental to a component is its interface that characterizes the functionality. Interface defines component's provided services and acts as a basis for its use and implementation. It acts as one of the primary definitive sources for understanding component and often may be the only available source. An interface comprises of set of operations which act as access points for interaction with the outside environment. However, it is important to note that an interface is simply a collection of operations and only describes these operations. An operation specifies how the inputs, outputs, and component states are related, and the effect of calling the operations on that relationship.

Table 1 NO/NP Complexity Metrics

NO\NP	1 – 19	20 – 50	51 +
1	Low	Low	Average
2 – 5	Low	Average	High
6 +	Average	High	High

Ideally, in an interface specification, the functional properties of a component are described. Function properties include a signature part in which the operations are described, and a behavior part, that addresses the overall behavior of a component. The interface signature spells out the individual elements of a component in a syntactic manner. In this section, we will consider the syntactic specification of a component while the semantic properties of a component are discussed in detail in section 3.2. In UML component specification, each interface has an associated interface information model. An Interface information model for a course registration component is shown as an example in Figure 1. It is a type model of the possible states of a component to which the operation specification can refer. Since this type model is at the specification level, it only specifies the states the component may have and does not describe the way in which the state is implemented or persisted [11].

In IFPUG [24] version of function point analysis [25] for object oriented systems, data functions are defined as the functionality provided to the user to meet internal and external data requirement; and are classified into two types (i) internal logical file (ILF) and (ii) external input file (EIF). The complexities of the ILF and EIF are determined by the data element type and the record element type. We present a measure of interface complexity similar to IFPUG function point count. However, in contrast to OO function point analysis [25], which is based on the data functions from the class diagrams, our metric is based on the UML interface information model that are available during the early stages of the CBS software development life cycle.

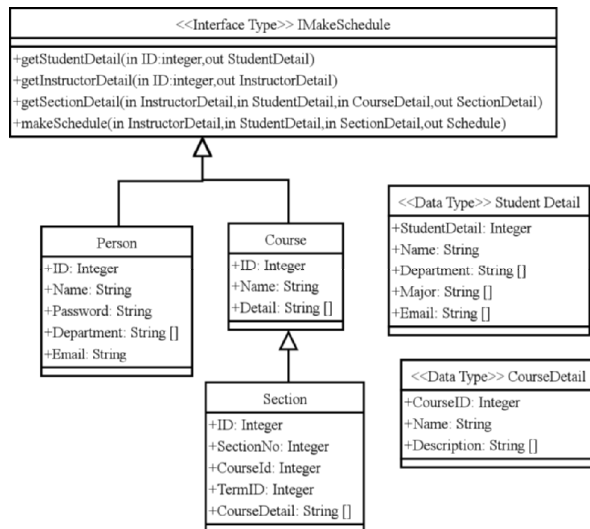


Figure 1 Interface Information Model

Based on UML component specification, we classify the interfaces that have some operations and exchange data with its environment as the candidate for function count. For each of the selected candidate, we determine its function type and classify them as either ILF or EIF. Interfaces that have operations which change the attributes of other interfaces in the exchange are regarded as ILF. All the remaining interfaces are categorized as EIF. We assume that an interface complexity increases with the larger number of operations and associated parameters. Further, functional complexities of both ILF and EIF is rated based on the number of operations (NO) and number of parameters (NP), NO/NP complexity metrics (Table 1). Finally, the count of each function type are classified according to their complexity and then assigned a weight using Table 2.

Table 2 Interface Complexity Metrics

Data Type	Low	Average	High	Total
ILFi	--- x 7 =	--- x 10 =	--- x 15 =	
EIFi	--- x 5 =	--- x 7 =	--- x 10 =	

Hence, the interface complexity measure of a component i , denoted by IC_i , is defined as

$$IC_i = ILFi + EIFi \quad (1)$$

where $ILFi$ and $EIFi$ are the weighted values for component interfaces classified based on their complexity.

3.2 Constraint

Component interface only spells out the individual elements of a component in mostly syntactic terms. However, components are subject to further constraints regarding their use. These constraints are both on individual elements as well as on the relationship among the elements. The pre-condition and post condition in terms of operation semantics and dependency of operation's invocation on another operation are the examples of such kind of constraints. It is important to have explicit specification of these constraints as they help in defining characteristics of a component. Further, it is also essential for a component user to understand these constraints for its proper and precise use [8].

In UML component specification, these constraints are specified precisely using OCL which is a declarative language that allows constructing logical expressions. The behavior of a component is determined by set of pre-conditions and post-conditions associated with each interfaces. Pre-conditions are the assertions the component assumes to fulfill before an operation is invoked. Post-conditions are the component guarantees will hold just after an operation has been invoked, provided the operation's pre-conditions were true when it was invoked. A pre-condition is, in general, a predicate over the operation's input parameters, while a post-condition is a predicate over both input and output parameters. Further, an operation's pre- and post-conditions will depend on the state maintained by the component. Thus, a set of invariants are also associated with an interface which is a predicate over the interface's state model that will always hold. Finally, a component specification holds a set of inter-interface conditions, which are predicates over the state model of all the component's interfaces [7, 11].

Pre-conditions, post-conditions and invariants of a constraint compose of one or more OCL clauses. Since OCL is a declarative language, each expression is written for an instance of specific type. For example '*Self*' is contextual instance that provides a reference point for interpreting an OCL expression. Similarly, other instances whose types are different to the type represented by the contextual instance are commonly accessed by OCL facilities like, variable definitions through 'Let' expression, 'if' expression condition,

predefined iterator variables, literals etc. The occurrence of each of these clauses, 'Self', 'Let', 'If', 'Sequence', 'Iterator', 'exists', '@pre' and 'Comparison', contribute to the constraint's complexity as they serve the same function as the predicates. Since OCL expressions are a set of sequential statements, we propose using McCabe's cyclomatic complexity [13] to measure component constraint complexity. In addition to obtaining complexity metrics from control flow graph, McCabe also observed that complexity of a program is equal to number of decision statements, called predicates, plus one for any program. Thus, the constraints complexity measure for each operation can be defined as

- (i) Number of predicates provided by UML/OCL specification, plus
- (ii) One

The constraints complexity measure of an interface j is represented as $V(GI_j)$. For each interface $j = 1, 2 \dots n$, the complexity measure for all the operation k ranging from $k = 1, 2 \dots n$, in an interface will be defined as

$$V(GI_j) = \sum_{k=1}^{\max} V(GO_k) \quad (2)$$

this is the sum of constraint complexity measure of all the operations in an interface; and 'max' is the total number of operations in an interface.

Thus, constraint complexity metrics for each component i will denoted as $V(GT_i)$ and defined as

$$V(GT_i) = \sum_{j=1}^n V(GI_j) \quad (3)$$

that is the sum of constraint complexity of all the interfaces $j = 1, 2, \dots n$; and 'n' represents the total number of interfaces in a component i .

Finally, the average constraints complexity metrics for a component i denoted by $A(GT_i)$ is defined as

$$A(GT_i) = V(GT_i) / NO_i \quad (4)$$

where NO_i is the total number of operations in the component i .

3.3 Interaction

Interaction among components is characterized by the use of a component's interface or through consuming other component's event. The interaction happens when a component provides an interface and other component uses it, and similarly when a component submits an event and other components receive it [10]. In UML component specification, collaboration diagrams are used to specify the interaction between components. Each collaboration diagram shows one or more interactions, where each interaction shows one possible execution flow. These component interaction diagrams can focus on one particular component and show how it uses the interfaces of other components; and they can be used to depict an extended set of interacting components in overall architecture. Interface information model that in detail specify all the operations and their associated constraints also helps in clarifying the definitions of component interactions [11, 26].

Component interaction is also a measure of the degree of interdependence between components. It is a well established principle the component coupling should be kept at an acceptable level to decrease its complexity, maintainability etc. Besides interaction frequency, content of each interaction also has a significant effect on overall interaction complexity. Thus, frequency of interactions between components and the information content of each interaction are two main contributory factors towards component interaction measurement.

We propose to measure frequency of interactions exchanged between the components by considering each component interface operation in turn and calculate number of interactions based on one or more corresponding collaboration diagrams. Each collaboration diagram shows one or more interactions, where each interaction shows one possible execution flow. Let M_o denote the number of messages exchanged among components in the execution of scenario S_x for an operation O_x . M_i denotes the set of all messages exchanged between the components active during the execution of scenario S_x for all operations. We define the interaction frequency IF_x for an operation O_x as a ratio of the number interactions exchanged in an operation O_x over the total number of interactions exchanged during scenario S_x .

$$IF_x = M_o / M_i \quad (5)$$

Similarly, the information content of an interaction is characterized in terms of information send and received by a component interface operation. In UML component specification, this information content range from primitive data types such as integers, strings etc to user defined structured data types and can be derived from operation signatures of the interfaces involved in an interaction. The structure of information content also has a significant effect on the overall component interaction. We propose to measure its complexity by representing it as a hierarchical directed graph called component data type graph, as shown in Figure 2. The node of a graph represents data entities and set of arcs represent the relationship between them. All the base level data types are primitive data types. The number of hierarchical levels and number of different data types are the two factors that contribute towards information content complexity.

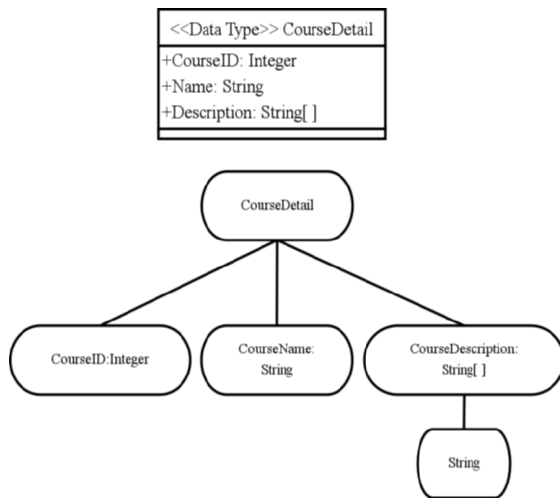


Figure 2 Component data type graph

The complexity of information content of an interaction can be measured using complexity metrics of a data structure graph proposed in [5]. The complexity of a directed graph W denoted by $CM(W, p)$, is used to measure the complexity of a data type W ; its value is defined as the sum of complexity of all its comprised data types, recursively calculated from the root type to its all primitive data types. The complexity metric of each data type is computed by the function,

$$CM(W, p) = p + \sum_{i=1}^n CM(Y_i, p+1) \quad (6)$$

where p is the number of the level where this data type occurs in the hierarchical graph and Y_i is a data type Y

of the i -th data field in data type including n data fields. For example, we measure the complexity of data type 'CourseDetail' to demonstrate the approach. The declaration and corresponding data type graph is shown in figure 2. The user defined data type 'CourseDetail' is information content containing three fields (CourseID, Name, and Description). 'CourseID' and 'Name' are fields of primitive data type's integer and string respectively. 'Description' is a data field containing array of strings. Thus, using equation 6, complexity of CourseDetail data type is given as:

$$\begin{aligned} CM(\text{CourseDetail}) &= 1 + CM(\text{CourseID}, 2) + \\ &\quad CM(\text{Name}, 2) + \\ &\quad CM(\text{Description}, 2) \\ &= 1 + CM(\text{Integer}, 2) + \\ &\quad CM(\text{String}, 2) + CM(\text{String}[], 3) \\ &= 1 + 2 + 2 + \{2 + CM(\text{String}, 3)\} \\ &= 1 + 2 + 2 + 2 + 3 = 10 \end{aligned}$$

Finally, we define the interaction complexity (CC) for a component as

$$CC = \sum_{i=1}^{\max} (IF_i \times \sum_{j=1}^n CM_j) \quad (7)$$

where ' i ' is the interface operation for a component and ' j ' is the number of data types involved in the information content exchange for the interface operation.

4. Conclusion and Future Work

In this paper, we have presented a new technique for measuring the complexity of a UML component specification. We have demonstrated that interface, constraints and interaction are the three primary factors that contribute to the complexity, and have described the metrics for measuring the complexity of each of these factors. These metrics provide a mechanism for estimating the complexity factor at specification level by considering the syntactic and semantic properties of a component, and the dynamic interaction behavior with other components. This technique enables the designers and developers to measure the component complexity by quantitative and objective metrics early in the specification phase to identify complex components, and consequently direct the appropriate amount of effort to develop and test these components to produce more reliable CBS.

In this work, we have gained several insights. We estimate collaboration diagrams complexity factor, which enables us to focus on the complex scenarios and use cases even though they may not be used often and yet they are reflected in the overall complexity measures. Our method can be used at the specification phase, thus being able to identify more complex components. Using the hierarchical directed graph model, we can also channel testing effort and resources appropriately. An important aspect of a component is its non-functional properties, such as performance and reliability. A limitation of UML component specification is that it does not support specification of these properties and the compositional and architectural aspects of CBS.

For future work, there is a need to assess the criticality of a component interaction as some component interactions can be more critical than others even though they might not be executed often; it is beneficial to estimate the distribution of the use cases complexity factor over different components, which allow us to make a list of critical use cases in the system. Further, to help reduce maintenance cost, a technique for measuring the maintainability of a CBD specification should be developed.

5. References

- [1] J. S. Davis and R. J. LeBlanc, "A study of the applicability of complexity measures," *IEEE Transactions on Software Engineering*, vol. 14, pp. 1366-1372, 1988.
- [2] E. J. Weyuker, "Evaluating Software Complexity Measures," *IEEE Transaction on Software Engineering*, vol. 14, pp. 1357-1365, 1988.
- [3] K. S. Lew, T. S. Dillon, and K. E. Forward, "Software complexity and its impact on software reliability," *IEEE Transactions on Software Engineering*, vol. 14, pp. 1645-1655, 1988.
- [4] J. Tian and M. V. Zelkowitz, "Complexity measure evaluation and selection," *IEEE Transactions on Software Engineering*, vol. 21, pp. 641-650, 1995.
- [5] S.-J. Huang and R. Lai, "Deriving Complexity Information from a Formal Communication Protocol Specification," *Software Practice and Experience*, vol. 28, pp. 1465-1491, 1998.
- [6] N. E. Fenton and N. Ohlsson, "Quantitative analysis of faults and failures in a complex software system," *IEEE Transactions on Software Engineering*, vol. 26, pp. 797-814, 2000.
- [7] I. Crnkovic and M. Larrson, *Building Component-based Reliable Software Systems*: Artech House, 2002.
- [8] J. Han, "A comprehensive interface definition framework for software components," In *Proceedings of Asia Pacific Software Engineering Conference*, pp. 110 - 117, 1998.
- [9] E. S. Cho, M. S. Kim, and S. D. Kim, "Component metrics to measure component quality," In *Proceedings of Eighth Asia-Pacific Software Engineering Conference*, pp. 419 - 426, 2001.
- [10] V. L. Narasimhan and B. Hendradjaya, "A New Suite of Metrics for the Integration of Software Components," In *Proceedings of Workshop on Object Systems and Software Architectures*, Victor Harbor, South Australia, 2004.
- [11] J. Cheesman and J. Daniels, *UML Components A Simple Process for Specifying Component Based Software*: Addison-Wesley, 2001.
- [12] V. Cote, P. Bourque, S. Oligny, and N. Rivard, "Software Metrics: An Overview of Recent Results," *Journal of Systems and Software*, vol. 8, pp. 121-131, 1988.
- [13] T. J. McCabe, "A Complexity Measure," *IEEE Transaction on Software Engineering*, vol. 2, pp. 308-320, 1976.
- [14] B. Henderson-Sellers, *Object-Oriented Metrics*: Prentice Hall, 1996.
- [15] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, pp. 476-493, 1994.
- [16] N. I. Churcher, M. J. Shepperd, S. Chidamber, and C. F. Kemerer, "Comments on "a metrics suite for object oriented design", " *IEEE Transactions on Software Engineering*, vol. 21, pp. 263-265, 1995.
- [17] V. B. Misic and D. N. Tesic, "Estimation of effort and complexity: An object-oriented case study," *Journal of Systems and Software*, vol. 41, pp. 133-143, 1998.
- [18] J. Verner and G. Tate, "Estimating Size and effort in fourth Generation Development," *IEEE Software*, vol. 5, pp. 15-22, 1988.

- [19] P. Bourque and V. Cote, "An Experiment in Software Sizing with Structural Analysis Metrics," *Journal of Systems and Software*, vol. 15, pp. 159-172, 1991.
- [20] M. H. Halstead, *Elements of Software Science*. New York: Elsevier, 1977.
- [21] A. Albercht and J. Gaffney, "Software Function, Source Lines of Code and Development Effort Prediction: A Software Science Validation," *IEEE Transaction on Software Engineering*, vol. 9, pp. 639 - 648, 1983.
- [22] S. Sedigh-Ali, A. Ghafoor, and R. A. Paul, "Software engineering metrics for COTS-based systems," *IEEE Computer*, vol. 34, pp. 44 - 50, 2001.
- [23] N. S. Gill and P. S. Grover, "Few important considerations for deriving interface complexity metric for component-based systems," *SIGSOFT Software Engineering Notes*, vol. 29, 2004.
- [24] IFPUG, *Function Point Counting Practices Manual, Release 4.1*: International Function Point Users Group, Princeton Junction NJ, 2000.
- [25] T. Uemura, S. Kusumoto, and K. Inoue, "Function Point Measurement Tool for UML Design Specification," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 13, pp. 223-243, 2001.
- [26] Y. Liu and H. C. Cunningham, "Mapping component specifications to Enterprise JavaBeans implementations," In *Proceedings of the 42nd annual Southeast regional conference*: ACM Press, pp. 177 - 182, 2004,.