

A complexity measure for UML component-based system specification



Sajjad Mahmood and Richard Lai^{*,†}

Department of Computer Science and Computer Engineering, La Trobe University, Victoria 3086, Australia

SUMMARY

A component-based system (CBS) is integration centric with a focus on assembling individual components to build a software system. In CBS, component source code information is usually unavailable. Each component also introduces added properties such as constraints associated with its use, interactions with other components and customizability properties. Recent research suggests that most faults are found in only a few system components. A complexity measure at a specification phase can identify these components. However, traditional complexity metrics are not adequate for a CBS as they focus mainly on either lines of code (LOC) or information based on object and class properties. There is therefore a need to develop a new technique for measuring the complexity of a CBS specification (CBSS). This paper describes a structural complexity measure for a CBSS written in Unified Modelling Language (UML) from a system analyst's point of view. A CBSS consists of individual component descriptions characterized by its syntactic, semantic and interaction properties. We identify three factors, interface, constraints and interaction, as primary contributors to the complexity of a CBSS. We also present an application of our technique to a university course registration system. Copyright © 2006 John Wiley & Sons, Ltd.

Received 5 May 2005; Revised 19 April 2006; Accepted 16 May 2006

KEY WORDS: component-based system; component-based system specification; UML; complexity; software metrics

1. INTRODUCTION

Software complexity is a key quality feature that has been measured widely in software systems [1–6] and has a proven impact on other software quality attributes such as maintainability, reliability and testability. Complexity assessment provides a useful means to identify potential troublesome software components that need more careful analysis and resource allocation. Since a component-based system

*Correspondence to: Richard Lai, Department of Computer Science and Computer Engineering, La Trobe University, Victoria 3086, Australia.

†E-mail: lai@cs.latrobe.edu.au

(CBS) constitutes assembling individual components in an interoperable manner, its complexity not only depends on individual components but also on associated interactions. The traditional complexity metrics are not adequate for a CBS because they are either dependent on lines of code (LOC) or measure complexity based on classes, objects and their inheritance properties. Further, information on a component's source code and class structure is usually unavailable. Added properties specific to a component are also introduced such as interface structure, associated constraints associated with its use, interactions among components, and customizability and reusability properties. Thus, the applicability of traditional metrics for a CBS is limited and a method is needed that adequately considers these properties during a CBS specification (CBSS) complexity measure.

Recent research suggests that most faults are found in only a few system's components [7]. If we can identify these components at an early specification phase, then precautionary actions can avoid the likelihood of failure and costly maintenance. A CBS complexity measure at a specification level provides an earlier quantitative assessment for more effective identification of fault-prone components. In this paper, we adopt Szyperski's *et al.* component definition [8]: 'A software component is a unit of composition with contractually specified interface and explicit context dependencies only. A software component can be deployed independently and is subject to composition by a third party'. Thus, we can characterize a CBSS, which consists of individual component definitions, by the component characteristics, context dependencies and the interaction between the other components.

The interface plays a fundamental role in defining a CBS based on a component's characteristics. The interface acts as a primary source for understanding, use and implementation, and spells out the individual elements of a component in a syntactic manner. In addition to syntactic specification, semantic information is necessary for a component's effective use. Such semantic information comprises a combination of the parameter values an operation accepts in order to constrain the order in which operations are invoked [9]. As well as interface and constraints which define the overall capability of a component, one needs a set of configurations for its proper use. This involves defining a component's role in a given context and its use in assorted contexts [10]. Thus, interaction is another important factor affecting overall complexity as it is a measure of the degree of interdependence between components.

So far, most CBS measurement research focuses on identifying CBS quality attributes and the limit of traditional measures when applied to a CBS. Cho *et al.* [11] proposed a suite of metrics based on the complexity information of all classes and their methods which comprises each component and captures the dynamic complexity based on analysis of a component source code. However, one of the limits of this technique is the lack of source-code information because of the black-box nature of components. Narasimhan and Hendradjaya [12] proposed component interaction complexity metrics by deriving packing density metrics and interaction density metrics. However, there is not much work done on deriving software metrics from a CBSS; in particular, there is little work done to measure the complexity of a CBSS based on its internal properties and external behaviour.

This paper describes a structural complexity measure for a CBSS written in Unified Modeling Language (UML) [13] from a system analyst's point of view. In this paper, we adopt structural complexity definition [14]: 'Structural complexity is that portion of overall (psychological) complexity that can be represented and thus assessed objectively'. We identify that for a CBSS, overall complexity is mainly attributed to interface, constraint and interaction. Our approach is distinct as it considers syntactic, semantic and interaction properties for a CBSS complexity measure. We also consider both interaction frequency and interaction content to calculate interaction complexity. Further, we present an

application of the proposed technique to a university course registration system. Our aim is to estimate CBSS complexity based on a quantitative metrics that can be evaluated early in a CBS life cycle. These metrics enable a system analyst to better understand the factors affecting complexity of a CBSS and provide a mechanism for identifying complex components. This is important as towards the latter stages of a CBS life cycle, complexity numbers can guide an analyst in the allocation of resources to the testing and maintenance phases.

2. RELATED WORK

Several complexity measurement models and estimation techniques have been proposed in the literature [3,14–19]. Some of the conventional metrics, such as LOC, need source-code analysis. Similarly, object-oriented metrics focus on classes, objects and their hierarchy properties. Traditional metrics put particular emphasis on measuring design aspects for a quality assessment in the early development phase. However, these traditional metrics [3,16,20–23] are not adequate for a CBS complexity measurement because of the lack of source-code information, measurement unit differences and scarce consideration of measurement factors.

Cho *et al.* proposed a suite of metrics for measuring the complexity, customizability and reusability of software component [11]. Component complexity metrics are defined as a combination of four types of metrics, namely, component plain complexity, component static complexity, component dynamic complexity and component cyclomatic complexity [11]. These metrics require the analysis of the complexity of each class and method. The metrics also require the analysis of component's source code which is rarely available. This makes it dependent on the component implementation methodology.

Interface and middleware code, needed for integrating different components, are the two main factors affecting the complexity of a CBS [24]. Gill and Grover have extended this idea by identifying interface complexity metrics based on an interface characterization model of a component. They identify the interface signature, interface constraints, interface packaging and configuration as factors contributing to the overall interface complexity of a component [25]. However, there is no discussion on the methodology for measuring and validating these identified attributes.

Narasimhan and Hendradjaya [12] 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 (CPD) and Component Interaction Density (CID) metrics. The CPD metrics relates a component to the number of integrated components and CID metrics relates interactions between components to the number of available interactions in the entire system. A limit of this technique is that it only considers one individual attribute (component interaction) as a means to measure the complexity of a component.

3. SPECIFICATION COMPLEXITY MEASURE

CBS departs from the conventional software systems as it is integration centric as opposed to development centric. In CBS, complexity not only depends on the individual components but also on the underlying architecture and the integration process. To measure the complexity of a UML CBSS, it is not practical to only consider one of the attribute of a CBS affecting its complexity.

Therefore, to have a reliable CBSS complexity measure, it is necessary to identify and appropriately measure in detail each factor affecting its complexity.

A CBSS consists of individual component descriptions characterized by its internal properties and external behaviour. Syntactic and semantic specification relates to internal properties. Interaction between components relates to external behaviour. We propose a structural complexity measure for a CBSS based on its individual components by considering the syntactic, semantic and interaction properties. After a detailed analysis, we have identified three factors, interface, constraints and interaction, as primary contributors to the complexity of a CBSS.

3.1. Interface

Fundamental to a component is its interface which characterizes the functionality provided. The interface defines the services provided by a component and acts as a basis for its use and implementation. It is one of the primary definitive sources for understanding a component and often can be the only source available. An interface comprises a set of operations that act as access points for an interaction with the outside environment. However, it is noteworthy that an interface is simply a collection of operations and only provides a description of them. An operation specifies how inputs, outputs and a component's state relate and the effect of calling the operations on that relationship.

Ideally, an interface specification describes the functional properties of a component. Function properties include a signature part, to describe the operations, and a behaviour part, to address the overall behaviour of a component. The interface signature delineates the individual elements of a component in a syntactic manner. In this section, we consider the syntactic specification of a component while the semantic property of a component is discussed in detail in Section 3.2. In UML component specification, each interface has an associated interface information model, an example is shown 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 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 [13].

In International Function Point User Group (IFPUG) [26] (an international standard—ISO/IEC 20926:2003), data functions are defined as a functionality provided to a user to meet internal and external data requirements, 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 (DET) and the record element type (RET). We present a measure of interface complexity based on the IFPUG function point count. We have selected the IFPUG version of function point analysis (FPA) because it is an international standard and has been applied at design specification phase [27] of a system development. However, in contrast to Uemura *et al.*'s function point analysis [27], which uses class and sequence diagrams as a source for object-oriented FPA, our metric is based on a UML interface information model that is available during the early stages of a CBS life cycle [9].

Based on the UML interface information model, we classify the interfaces that have some operations and also exchange data with their environment as candidates for function count. We classify each of the selected candidates as either *ILF* or *EIF*. Interfaces that have operations that change the attributes of other interfaces in the data exchange are classified as *ILF*. All of the remaining interfaces are classified as *EIF*.

In IFPUG, each identified *ILF* and *EIF* is ranked, based on the number of DET and RET, using RET/DET metrics [26,27]. Since RET is a user recognizable subgroup, we count the number of

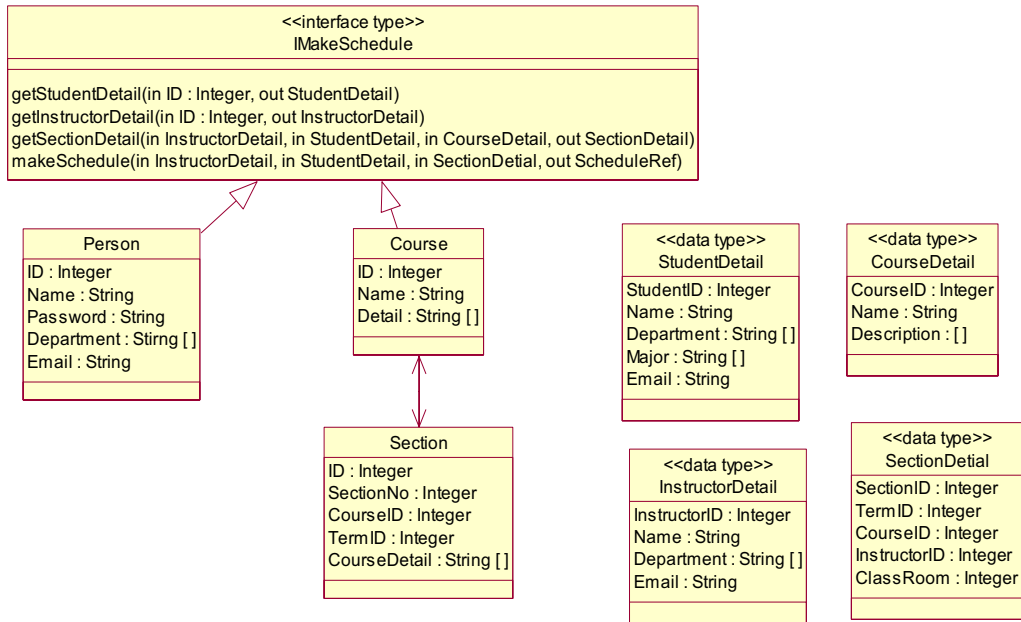


Figure 1. Interface information model for 'IMakeSchedule'.

Table I. NO/NP complexity metrics.

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

operations (NO) in an interface. Similarly, DET is a unique user recognizable field; we count the number of parameters (NP) in an interface. By analogy with RET/DET metrics [26], we propose NO/NP complexity metrics, shown in Table I, to rank candidate interface. Ranked interfaces are assigned weights based on IFPUG standard weights, shown in Table II.

Finally, we define an interface complexity measure of a component i , denoted by IC_i , as

$$IC_i = \sum_{i=1}^n ILF_i + \sum_{j=1}^n EIF_j \quad (1)$$

Table II. Component interface complexity metrics.

Data type	Low	Average	High
ILF_i	$— \times 7 =$	$— \times 10 =$	$— \times 15 =$
EIF_i	$— \times 5 =$	$— \times 7 =$	$— \times 10 =$

Where ILF_i and EIF_i are the weighted values for a component interface classified based on its complexity.

3.2. Constraint

A component interface only defines the individual elements of a component mostly in syntactic terms. However, a component is subject to further constraints on its 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 an operation semantics and dependency of an operation's invocation on another operation are examples of constraints. It is important to have an explicit specification of these constraints as they are useful for defining the characteristics of a component. Similarly, it is essential for a component user to understand the constraints to enable its proper and precise use [10].

UML CBSS uses the Object Constraint Language (OCL), a declarative language that allows logical expressions to be composed, to identify constraints as sets of pre-conditions and post-conditions associated with each interface. Pre-conditions are the assertions a component assumes to fulfil before an operation is invoked. Post-conditions are a component guaranteed to hold after an operation has been invoked, provided that the operation's pre-conditions were true when it was invoked. A pre-condition is, in general, a predicate over an operation's input parameters, while a post-condition is a predicate over both input and output parameters. Further, an operation's pre-conditions and post-conditions will depend on the state maintained by a component. Thus, a set of invariants associate with an interface 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 of the component's interfaces [9,13].

Pre-conditions, post-conditions and invariants of a constraint comprise one or more OCL clauses. Since OCL is a declarative language, each expression is written in the context of an instance of a specific type. '*Self*' is a contextual instance that provides a reference point for interpretation of an OCL expression. Similarly, other instance types, different to the type represented by a contextual instance, are commonly accessed by OCL functions such as variable definitions through 'Let' expression, 'if' expression condition, predefined iterator variables, etc. Occurrences of OCL clauses 'Let', 'If', 'ForAll', 'isUnique', 'Exists', 'Self', '@pre' and 'Iterate' contribute to the constraint's complexity. They serve the function of a predicate. Since OCL expressions are a set of sequential statements, we propose using the McCabe's cyclomatic complexity [16] to measure constraint complexity. According to McCabe, the complexity of a program is equal to the number of decision statements, called predicates, plus one for any program. Thus, we define the constraints complexity measure for each operation as the number of predicates provided by UML/OCL specification, plus one.

Further, we consider complexity of each predicate to be one because the counted OCL clauses are binary decision statements (e.g. ‘IF’, ‘Exists’) and according to the McCabe’s cyclomatic complexity, a binary decision statement contributes a complexity count of one.

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

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

that is, the sum of the 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 the 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 number of operations in a component i .

3.3. Interaction

Interaction between components is characterized by a component’s interface or through using other component’s event. The interaction occurs when a component provides an interface and another component uses it. Similarly, when a component submits an event and other components receive it [12]. In UML CBSS, collaboration diagrams specify the interactions 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 reveal how it uses the interfaces of other components; and they depict an extended set of interacting components in an architecture. An interface information model specifies all of the operations and their associated constraints. It also assists in clarifying the definitions of component interactions [13,28].

Component interaction is also a measure of the degree of interdependence between components. It is a well-established principle that component coupling should be kept to a minimum to decrease its complexity, maintainability, etc. The frequency of interactions between components and the information content of each interaction are the main contributory factors to a component interaction measurement.

We extend the concept of *normalized dynamic coupling of a connector* presented in [29], to define the frequency of interactions exchanged between the components, by considering each component interface operation in turn and calculating the number of interactions based on one or more corresponding collaboration diagram. Each collaboration diagram shows one or more interaction,

where each interaction shows one possible execution flow. Let M_o denote the number of messages exchanged among components for an operation O_x in interface I_I . Let M_I denote the set of all messages exchanged between the components for all operations in interface I_I . We define the interaction frequency IF_x for an operation O_x in interface I_I as a ratio of the number of interactions exchanged in an operation O_x over the total number of interactions exchanged in an interface I_I :

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

The information content of an interaction contributes to the complexity of a CBSS because the components interact with one another through shared information. Information content also plays a significant role in the decision making within a component. The overall complexity of interaction can therefore be characterized by the type and amount of information passed between them. The information content of an interaction is characterized by information sent and received by a component interface operation. In UML CBSS, this information content ranges from primitive data types such as integers, strings, etc., to user-defined structured data types and can be derived from operation signatures of interfaces involved in an interaction.

The structure of information content has a significant effect on the overall component interaction. We propose to measure its complexity by representing information content as a hierarchical directed graph called a component data type graph, as shown in Figure 2. The node of the graph represents data entities and a set of arcs represent the relationships between them. All of the base-level data types are primitive data types. The number of hierarchical levels and number of different data types are the two factors which contribute towards information content complexity. We propose to measure information content complexity by adopting the coupling complexity metrics presented in one of the author's previous work [6]. 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 a sum of the complexity of all of its comprised data types, recursively calculated from the root type to all of its 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 of an i th data field in a data type which includes n data fields. For example, we measure the complexity of data type 'CourseDetail' to explain 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, CourseName and CourseDescription). CourseID and CourseName are fields of a primitive data type's integer and string, respectively. CourseDescription is a data field containing array of strings. Thus, using Equation (6), the complexity of CourseDetail data type is given as:

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

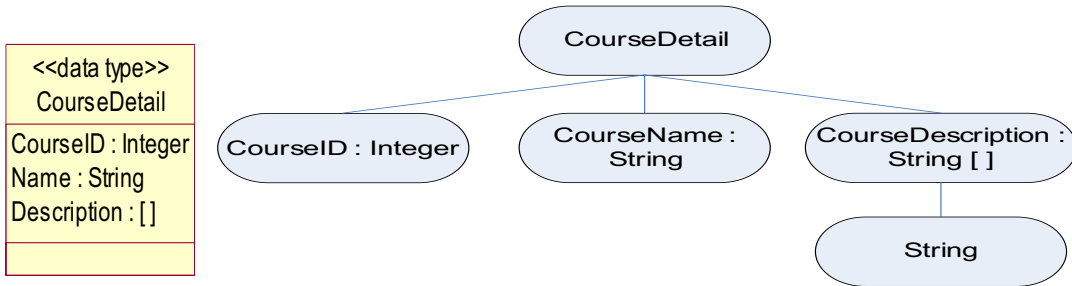


Figure 2. Component data type graph for 'CourseDetail'.

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

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

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

4. AN APPLICATION

This section discusses an application of our proposed metrics to measure the complexity of a UML CBSS. The University Course Registration System (UCRS) [28] is used to measure the complexity of a UML CBSS, and show the usefulness of the work. Within this system, a student registers for classes. Once given access, the students may select a term and build a class schedule from the offered classes. The system passes information about a student's schedule to the billing system. A student can also register, add and drop a course. An instructor may use the registration system to print a student class list and to submit grades for her/his class. The administrator may maintain student and teacher information.

Figure 3 shows the business type model derived from the component identification stage. It identifies Person, Course and Term as core types. Based on these three core types, we further divide the system into four components. The 'PersonManagement' component includes the Person, Student, Instructor, 'StudentSchedule' and 'InstructorSchedule'. The 'CourseManagement' component includes types 'Course' and 'Section'. The 'Term Management' component only includes 'Term'. The component architecture diagram in Figure 4 shows the main components required by the system. We keep the Billing component as a separate entity as we assume that it is supplied by a separate billing system. These diagrams provide an overall view of the system and help to demonstrate our proposed specification metrics.

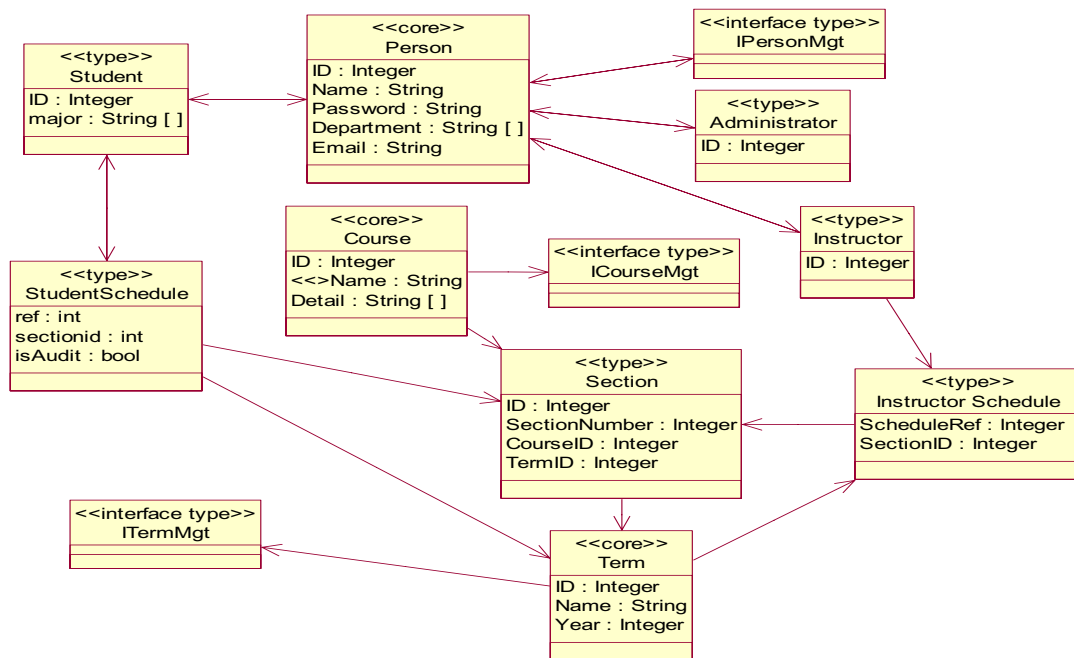


Figure 3. Interface type model for UCRS.

4.1. Interface

For the component ‘RegistrationSystem’, as shown in Figure 4, we have seven interfaces namely, IMakeSchedule, IUpdateSchedule, IDisplaySchedule, IRegisterCourse, IViewResult, ISubmitGrades and ILogin. From the above-mentioned interfaces, IDisplaySchedule, IViewResult and ILogin qualify as *EIF* because the operations of these interfaces do not change the attributes of other interfaces in exchanging the information. The remaining four interfaces are classified as *ILF* because they affect the attributes of other interfaces in the system during the message exchange. Next, the functional complexity of these interfaces is calculated. For example, Figure 1 shows the interface information model of ‘IMakeSchedule’. In this interface, there are four operations and 12 associated parameters, thus, based on Table I, ‘IMakeSchedule’ interface’s functional complexity is low. Similarly, we can calculate the functional complexity of all ‘RegistrationSystem’ interfaces, which is ‘Low’ in this example. Finally, these functional complexities are assigned weights, based on Table II, and the sum of these weighted functional complexity values reflect the interface measure. Since both the *ILF* and *EIF* type of ‘RegistrationSystem’ components have low complexity, their weighted complexity values will be ($4 \times 7 = 28$) and ($3 \times 5 = 15$), respectively. Therefore, the interface measure

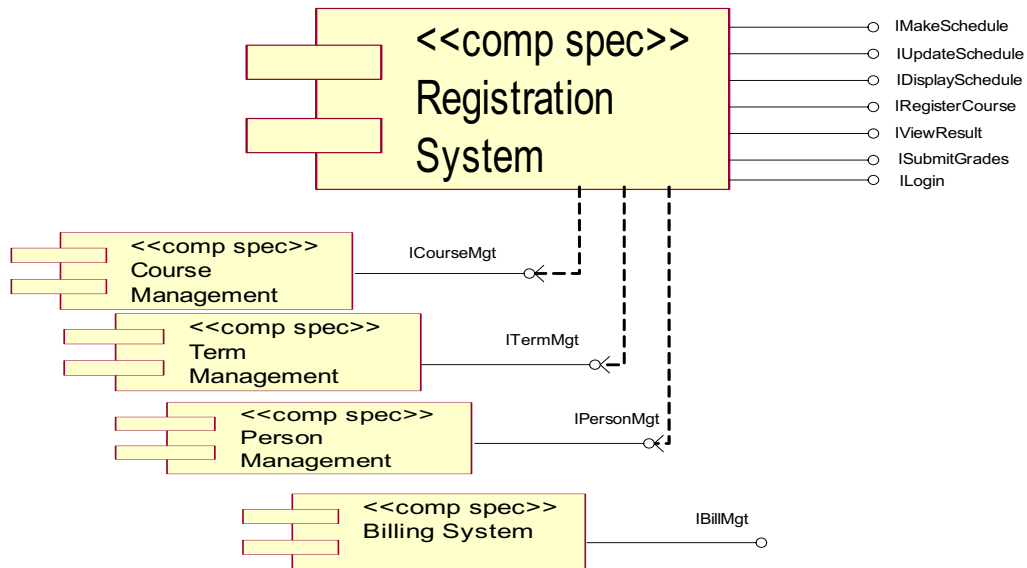


Figure 4. Component specification architecture.

Table III. Interface complexity metrics of UCRS.

Component	<i>ILF</i>	<i>EIF</i>	<i>IC</i>
RegistrationSystem	28	15	43
PersonManagement	21	12	33
CourseManagement	14	5	19
TermManagement	14	5	19

of 'RegistrationSystem' component is 43, which is the sum of the weighted functional complexity of *ILF* and *EIF*. Similarly, the interface complexity measure for the other three components can be easily computed, as shown in Table III.

4.2. Constraint

In the 'PersonManagement' component, there are 10 individual operations, as shown in Figure 5. Each operation has a set of OCL constraints associated with them. To illustrate our constraints metrics, we first select the 'addNewStudent' operation. There are five occurrences of OCL predicates ('Exits', 'Let', 'If') among the eight clauses group defined in Section 3.2. Based on Equation (2), the constraints complexity of 'addNewStudent' is 6.

IPersonMgt
addNewStudent(in student : StudentDetail) : Boolean updateStudentDetail(in ID : StudentID, out student : StudentDetail) : Boolean getStudentDetail(in ID : StudentID, out StudentDetail) makeStudentSchedule(in student : StudentDetail, in section : SectionDetail, out schedule : ScheduleDetail) updateStudentSchedule(in ID : StudentID, in section : SectionDetail, in operation "add" or "drop", out schedule : ScheduleDetail) addNewInstructor(in instructor : InstructorDetail) : Boolean updateInstructorDetail(in ID : InstructorID, out instructor : InstructorDetail) : Boolean getInstructorDetail(in ID : InstructorID, out InstructorDetail) makeInstructorSchedule(in ID : InstructorID, in section : SectionDetail, out schedule : ScheduleDetail) updateInstructorSchedule(in ID : InstructorID, in section : SectionDetail, in operation "add" or "drop", out schedule : ScheduleDetail)

Figure 5. 'PersonManagement' component operations.

IPersonMgt : makeStudentSchedule (in student: StudentDetail, in section: SectionDetail,
out schedule : ScheduleDetail)

Pre:

----- student and section information are valid -----
 Student - > **exists** (s | id = student.StudentID) and
 Section - > **exists** (sec | id = section.SectionID) and

Post:

StudentSchedule @pre -> **ForAll** (sr | sr.scheduleRef <> schedule.scheduleRef) and

Let s = (StudentSchedule @pre → asSequence → first in
 s. schedule .scheduleRef = schedule.scheduleRef and
 s. schedule.id = schedule.id and
 schedule.id = StudentDetail.StudentID and
 s.schedule.section = schedule.section and
 schedule.section = sectionDetail.sectionID

Figure 6. 'makeStudentSchedule' OCL specifications.

Similarly, 'makeStudentSchedule', as shown in Figure 6, has six occurrences of OCL predicates ('Exists', '@pre', 'ForAll', 'Let') and its constraints complexity is 7. In the same way, the constraints complexity of other operations is calculated, and their respective values are added together to obtain the constraints measure for the component. In our example, the constraint measure for the 'PersonManagement' component is 56. Since the total number of operations is 10, the average constraints complexity is $(A(GT) = 56/10 = 5.6)$. Similarly, the constraints complexity of the other three components can easily be computed, as shown in Table IV.

Table IV. Constraints complexity of UCRS.

Component	$V(GT)$	$A(GT)$
RegistrationSystem	85	5
PersonManagement	56	5.6
CourseManagement	40	5.1
TermManagement	35	4.4

Table V. Information content of data types in UCRS.

Data type	CM
CourseDetail	10
StudentDetail	17
ScheduleDetail	22
TermDetail	7
SectionDetail	11
InstructorDetail	12

4.3. Interaction

Interaction complexity is based on the interaction frequency and information content of each interaction. First, the information content of all parameters used in UCRS is calculated. The ‘StudentDetail’ is a user-defined data type which consists of five data elements, namely, StudentID, Name, Department, Major and Email. The data type Name and Email are of string data type while StudentID is integer. The data types of Major and Department are a set of string arrays. Therefore, the information content complexity of ‘StudentDetail’ can be calculated, based on Equation (6), as follows:

$$\begin{aligned}
 CM(\text{StudentDetail}) &= 1 + CM(\text{StudentID}, 2) + CM(\text{Name}, 2) + CM(\text{Department}, 2) \\
 &\quad + CM(\text{Major}, 2) + CM(\text{Email}, 2) \\
 &= 1 + CM(\text{Integer}, 2) + CM(\text{String}, 2) \\
 &\quad + \{2 + CM(\text{String}[, 3])\} + \{2 + CM(\text{String}[, 3])\} + CM(\text{String}, 2) \\
 &= 1 + 2 + 2 + \{2 + CM(\text{String}, 3)\} + \{2 + CM(\text{String}, 3)\} + 2 \\
 &= 1 + 2 + 2 + 2 + 2 + 3 + 3 + 2 = 17
 \end{aligned}$$

Data structure information content of all of the data types in UCRS are shown in Table V.

Figure 7 shows the set of collaboration diagrams associated with the ‘IMakeSchedule’ interface of the registration component. The ‘IMakeSchedule’ interface has four operations namely,

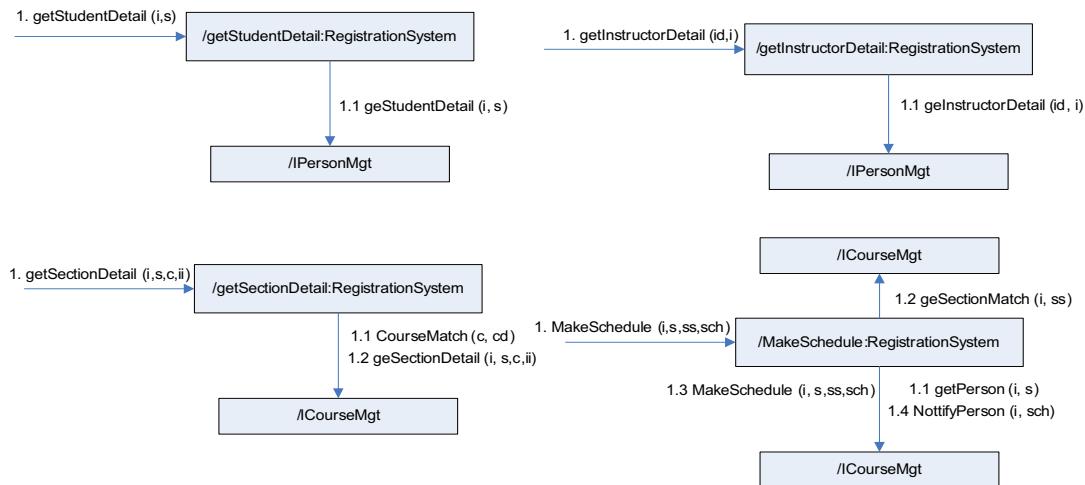


Figure 7. Collaboration diagrams of the 'IMakeSchedule' Interface.

Table VI. Interaction Measure for 'IMakeSchedule' interface.

Operation	IF_i	CM_i	CC_i
getStudentDetail	$1/8 = 0.125$	17	2.125
getInstructorDetail	$1/8 = 0.125$	12	1.5
getSectionDetail	$2/8 = 0.25$	$12 + 10 + 11 = 33$	8.25
makeSchedule	$4/8 = 0.5$	$12 + 11 = 23$	11.5
$CC \text{ IMakeSchedule}$			23.375

'getStudentDetail', 'getInstructorDetail', 'getSectionDetail' and 'makeSchedule'. The number of messages exchanged for interactions of both 'getStudentDetail' and a 'getInstructorDetail' operation is 1. Similarly, the number of messages exchanged by 'getSectionDetail' and 'makeSchedule' operations is 2 and 4, respectively. Thus, the interaction frequency of 'IMakeSchedule' interface will be 8, which is the sum of all of the exchange of messages associated with the interface. As shown in Figure 1, the information content types associated with the interface are 'StudentDetail', 'InstructorDetail', 'CourseDetail' and 'SectionDetail'. Therefore, we calculate the interaction factor for the interface based on Equation (7), as shown in Table VI.

The interaction complexity metrics of all of the interfaces in the 'RegistrationSystem' component are shown in Table VII. Similarly, all interaction complexity for all the components in UCRS can also be calculated along the same lines as that of 'RegistrationSystem' component. The interaction complexity metrics of all of the components in the system are shown in Table VIII.

Table VII. Interaction measure for the 'RegistrationSystem' component.

Interface	Total interactions	CC_i
IMakeSchedule	8	23.37
IUpdateSchedule	7	31.39
IDisplaySchedule	5	30.4
IRegisterCourse	15	25.94
IViewResult	6	9.65
ISubmitGrades	6	13.24
ILogin	2	8.75

Table VIII. Interaction measure for UCRS.

Component	CC
RegistrationSystem	143.34
PersonManagement	103.97
CourseManagement	67.02
TermManagement	55.12

Table IX. Component complexity.

Component	IC	$V(GT)$	CC
RegistrationSystem	43	85	143.34
PersonManagement	33	56	103.97
CourseManagement	19	40	67.02
TermManagement	19	35	55.12

5. RESULTS

Table IX shows the complexity measure of three primary factors for each component in UCRS. It indicates that the 'RegistrationSystem' component is the most complex, based on its interfaces, constraints and interaction factors. The 'PersonManagement' is the second most complex component. The results provide the quantitative measure of each component's syntactic and semantic properties, and its interaction with other component in UCRS.

Further analysis of the results of complexity measure for UCRS helps in identifying some important complexity information shown in the following.

- (a) From the interface measure of each component, as shown in Table III, all of the components can be classified into two complexity categories, the components 'RegistrationSystem' and 'PersonManagement' are of the same group with the higher interface complexity; the other components 'CourseManagement' and 'TermManagement' are less complex in their interfaces.
- (b) From the constraints complexity measure, shown in Table IV, the component 'RegistrationSystem' has the highest complexity ($V(GT) = 85$) and it has the highest number of operations. However, average constraints complexity ($A(GT) = 5$) is less than the 'PersonManagement' component. It shows that each operation in the 'RegistrationSystem' component has fewer constraints as compared with the 'PersonManagement' component.
- (c) The 'PersonManagement' component has the highest average constraints complexity ($A(GT) = 5.6$). This shows that a single operation in the component has the highest constraints complexity. The 'RegistrationSystem' and 'CourseManagement' components have almost the same average constraints complexity, which shows that although two components may have a different number of operations, they can still have the same constraints complexity.
- (d) The complexity measure of component interaction, shown in Table VII, shows that the interfaces 'IUpdateSchedule' and 'IDisplaySchedule' are more complex even though the number of their interactions is less than other interfaces in the component. This indicates that each interaction involving these interfaces has higher complexity of information content on its interaction with its environment. When comparing these two interfaces, 'IUpdateSchedule' is more complex than the interface 'IDisplaySchedule'.
- (e) As shown in Table V, the hierarchical tree structure of all of the information contents in UCRS is not complex. The number of levels of hierarchical tree is three and only the 'ScheduleDetail' has four levels. For the number of peer data types at the same level, the data types within the UCRS specification have values of 3–4.

Thus, based on our proposed metrics, we can identify complex components and associated interactions early in the specification phase of a CBS life cycle. These complexity numbers enable a system analyst to identify and make quantitative decisions in identifying faulty components. Further, towards the end of a CBS life cycle, these complexity numbers can guide system analysts to where they should concentrate their testing efforts.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a new technique for measuring the complexity of a UML CBSS. We have identified that interface, constraints and interaction are the three primary factors that contribute to the complexity, and have presented a set of metrics for measuring their complexity. These metrics provide a mechanism for estimating the complexity at a specification level by considering the syntactic and semantic properties of a component, and the dynamic interaction behaviour with other components. We also show the usefulness of these metrics by applying them to UCRS. This technique enables a system analyst to measure CBSS complexity by quantitative and objective metrics early in the specification phase, which helps in identifying complex components. Thus, a system analyst can direct an appropriate amount of effort to test and maintain these components to enable the production of a reliable CBS.

In this work, we have gained a few insights. We have estimated the collaboration diagram's complexity factor, which enables us to focus on the complex scenarios even though they may not be used often and yet they reflect in the overall complexity measures. Our method can be used at a specification phase, thus being able to identify more complex components. Using the hierarchical directed graph model, we can channel testing efforts and resources appropriately. An important aspect of a CBSS is its non-functional properties, such as performance and reliability. A limitation of current CBS UML specification is that it does not support the representation of the non-functional properties and architectural aspects of a CBS.

For future work, there is a need to assess how critical interactions are and estimate the distribution of use case complexity over different components. Complexity measures can be used to classify components based on their inherent characteristics. We plan to automate the collection of our metrics since they are based on a machine readable UML specification. Further, to help reduce maintenance cost, a technique for measuring the maintainability of a CBS should be developed [30].

REFERENCES

1. Briand LC, Morasca S, Basili VR. Property based software engineering measurement. *IEEE Transactions on Software Engineering* 1996; **22**(1):68–86.
2. Davis JS, LeBlanc RJ. A study of the applicability of complexity measures. *IEEE Transactions on Software Engineering* 1988; **14**(9):1366–1372.
3. Weyuker EJ. Evaluating software complexity measures. *IEEE Transactions on Software Engineering* 1988; **14**(9):1357–1365.
4. Lew KS, Dillon TS, Forward KE. Software complexity and its impact on software reliability. *IEEE Transactions on Software Engineering* 1988; **14**(11):1645–1655.
5. Tian J, Zelkowitz MV. Complexity measure evaluation and selection. *IEEE Transactions on Software Engineering* 1995; **21**(8):641–650.
6. Huang S-J, Lai R. Deriving complexity information from a formal communication protocol specification. *Software: Practice and Experience* 1998; **28**(14):1465–1491.
7. Fenton NE, Ohlsson N. Quantitative analysis of faults and failures in a complex software system. *IEEE Transactions on Software Engineering* 2000; **26**(8):797–814.
8. Szyperski C, Gruntz D, Murer S. *Component Software—Beyond Object-Oriented Programming* (2nd edn). Addison-Wesley: Reading, MA, 2002.
9. Crnkovic I, Larsson M. *Building Component-based Reliable Software Systems*. Artech House: Norwood, MA, 2002.
10. Han J. A comprehensive interface definition framework for software components. *Proceedings of the 1998 Asia Pacific Software Engineering Conference*, Taipei, Taiwan, 1998. IEEE Computer Society Press: Los Alamitos, CA, 1998; 110–117.
11. Cho ES, Kim MS, Kim SD. Component metrics to measure component quality. *Proceedings of the 8th Asia-Pacific Software Engineering Conference*, Macao, China, 2001. IEEE Computer Society Press: Los Alamitos, CA, 2001; 419–426.
12. Narasimhan VL, Hendradjaya B. A new suite of metrics for the integration of software components. *Technical Report (The 1st International Workshop on Object Systems and Software Architectures)*, University of Adelaide, Australia, 2004.
13. Cheesman J, Daniels J. *UML Components A Simple Process for Specifying Component Based Software*. Addison-Wesley: Reading, MA, 2001.
14. Henderson-Sellers B. *Object-Oriented Metrics—Measures of Complexity*. Prentice-Hall: Englewood Cliffs, NJ, 1996.
15. Cote V *et al.* Software metrics: An overview of recent results. *Journal of Systems and Software* 1988; **8**(2):121–131.
16. McCabe TJ. A complexity measure. *IEEE Transaction on Software Engineering* 1976; **2**(4):308–320.
17. Chidamber SR, Kemerer CF. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering* 1994; **20**(6):476–493.
18. Churcher NI, Shepperd MJ, Chidamber S, Kemerer CF. Comments on a metrics suite for object oriented design. *IEEE Transactions on Software Engineering* 1995; **21**(3):263–265.
19. Misisic VB, Tesic DN. Estimation of effort and complexity: An object-oriented case study. *Journal of Systems and Software* 1998; **41**(2):133–143.

20. Verner J, Tate G. Estimating size and effort in fourth generation development. *IEEE Software* 1988; **5**:15–22.
21. Bourque P, Cote V. An experiment in software sizing with structural analysis metrics. *Journal of Systems and Software* 1991; **15**(2):159–172.
22. Halstead MH. *Elements of Software Science*. Elsevier: New York, 1977.
23. Albercht A, Gaffney J. Software function, source lines of code and development effort prediction: A software science validation. *IEEE Transactions on Software Engineering* 1983; **9**(6):639–648.
24. Sedigh-Ali S, Ghafoor A, Paul RA. Software engineering metrics for COTS-based systems. *IEEE Computer* 2001; **34**(5):44–50.
25. Gill NS, Grover PS. Few important considerations for deriving interface complexity metric for component-based systems. *SIGSOFT Software Engineering Notes* 2004; **29**(2):4.
26. IFPUG. *Function Point Counting Practices Manual, Release 4.1*. International Function Point Users Group: Princeton Junction, NJ, 2000.
27. Uemura T, Kusumoto S, Inoue K. Function point measurement tool for UML design specification. *Journal of Software Maintenance and Evolution: Research and Practice* 2001; **13**:223–243.
28. Liu Y, Cunningham HC. Mapping component specifications to Enterprise JavaBeans implementations. *Proceedings of the 42nd Annual Southeast Regional Conference, 2004*. ACM Press: New York, 2004; 177–182.
29. Goseva-Popstojanova K, Hassan A, Guedem A, Abdelmoez W, Eldin D, Nassar M, Ammar H, Mili A. Architectural-level risk analysis using UML. *IEEE Transactions on Software Engineering* 2003; **29**(10):946–960.
30. Huang S-J, Lai R. Measuring the maintainability of a communication protocol based on its formal specification. *IEEE Transactions on Software Engineering* 2003; **29**(4):327–344.