A Complexity Metric for Black Box Components

Navneet Kaur, Ashima Singh

Abstract*—The Component Based Software Development (CBSD) approach is becoming the trend for software development which is based on developing the software from existing components instead of developing software from scratch everytime. Measuring software complexity is an important aspect during software development. Because software complexity is an important determinant of software development effort, testing effort , cost, maintainability etc. Researchers have proposed a wide range of complexity metrics for software systems . But the traditional software product and process metrics are neither suitable nor sufficient in measuring the Component and Component Based Software (CBS) complexity. So CBSD provides one of the central problems in measuring component and CBS complexity. Measuring component complexity plays an important role in determining the complexity of CBS system. Because component complexity affects the complexity of whole CBS . Component complexity affects integration and testing effort, cost, maintainability of CBS system . But now a days black box components are being used during CBSD and most of the time source code is not available which creates difficulty in measuring component complexity. In this paper a metric has been proposed for determining the black box component complexity. The proposed metric measures component complexity on the basis of component interface specification and use the concept of assigned weights.

Index Terms*— Black Box Component, CBSD, CBS system , component complexity, complexity metrics, traditional software product and process metrics.

I. INTRODUCTION

The Component Based Software Development(CBSD) approach is increasingly being adopted for software development. This approach uses reusable components as building blocks for constructing software systems. CBSD provides advantages like reduced development time, cost and effort, increased quality along with many others. These advantages are mainly provided by the reuse of already built-in software components. The following Fig.1 shows the technique for developing software from existing components.

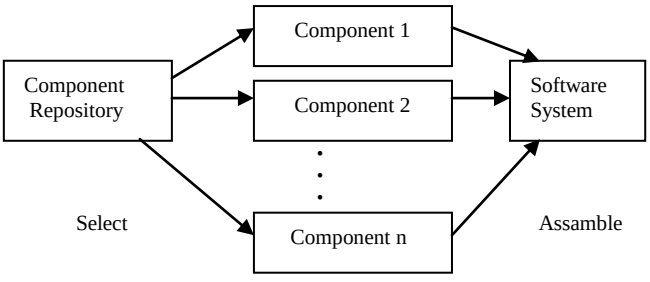

Fig .1 Component based Software development technique

Manuscript received on May, 2013.

Navneet Kaur, Computer Science & Engineering Department, Thapar University, Patiala, India.

Ashima Singh, Computer Science & Engineering Department, Thapar University, Patiala, India.

But it is neccessary to measure the software complexity in each software development approach because software complexity affects software development effort,cost, testability,maintainability etc. So many metrics have been proposed for measuring software complexity. But traditional software product and process metrics are not sufficent for measuring the component and Component Based Software (CBS) complexity. So CBSD provides one of the central problems in measuring component and CBS complexity. Measuring component complexity plays an important role in determining CBS system complexity. Because component complexity affects the complexity of whole CBS system .The component complexity is an important factor affecting the integration complexity, understandability,testability, maintainability etc of CBS system . But now a days black box components are being provided by component vendors for reuse and most of the time source code is not provided with components which creates difficulty in measuring component complexity. In this paper a complexity metric for black box component, CCM(BB) , has been proposed. The proposed metric is based on the component interface specification and use the concept of assigned weights.

The CBS system complexity is mainly calculated on the basis of it's components complexity . Thus by measuring the component complexity and selecting the less complex component during the component selection, the whole complexity of CBS system can be reduced. Like complex components will increase the integration effort(glue code complexity) ,testing effort and maintenance effort etc

II. BRIEF DISCUSSION OF SOME EXISTING METRICS

In this section some existing object oriented metrics and component integration metrics have been discussed.

A. Object Oriented Metrics

There are many object oriented metrics that can be used to measure the component based software complexity. Some object oriented metrics have been discussed below :

Metric 1: Weighted Methods Per Class (WMC)

This metric gives the combined complexity of local methods in a given class. The greater value of this metric shows more complexity, increase in testing effort and decrease in understandability.

Metric 2: Depth of Inheritance (DIT)

This metric is for class . It gives maximum length from the class node to root. More length means more complexity.

Metric 3: Response For Class (RFC)

The RFC metric gives the number of methods that can execute in response to a message sent to an object with in this class ,using to one level of nesting.

Metric 4: Coupling Between Objects (CBO)

For a given class, this metric measures the number of other classes to which the class is coupled. High value of this metric

shows the poor design, difficulty in understanding, decrease in reuse and increase in maintenance effort.

Metric 5: Lack of Cohesion Method (LCOM)

The cohesion of a class is characterized by how closely the local methods are related to the local instance variables in the class. LCOM is defined as the number of disjoint sets of local methods. High value of this metric shows good class subdivision.

Metric 6: Number of Children (NOC)

This metric is based on a node (class) of inheritance tree. It gives the number of immediate successors of the class. High value of this metric shows more reuse, poor design and increase in testing effort.

Metric 7: Lines of Code (LOC)

This metric is based on the size of methods. It gives measure of physical lines , statements , and/or comments. High value of this metric shows more complexity .

Metric 8: Cyclomatic Complexity (CC)

This metric measures the complexity of methods. It gives the measure of independent algorithmic test paths. More independent paths means more testing effrot.

B. Metrics for the Integration of Software Components

a) Metric 1: Component Packing Density (CPD)

The CPD metric measures the component constituents to the number of integrated components. This metric is used to identify the density of integrated components. Thus, a higher density represents a higher complexity.

$$CPD_{<constituent\_type>} = \frac{\#<Constituent>}{\#\ Components}$$

Where #<Constituent> is the number of lines of code, operations, classes, and/or modules in the related components.

b) Metric 2: Component Interaction Density (CID)

The CID metric measures the ratio of actual number of interactions to the available number of interactions in a component.

$$CID = \frac{\#I}{\#\ Imax}$$

Where #I and #Imax represents the number of actual interactions and maximum available interactions respectively . If one component provides interface and another components use it or if one component submits an event and another component receive it, then it is called an interaction. When the density of interaction increases, complexity increases.

Metric 3: Component sIncoming Interaction Density (CIID)

The CIID metric measures the ratio of actual number of incoming interactions to the maximum available incoming interactions in a component.

$$CIID = \frac{\#\ Iin}{\#\ Imax\_in}$$

Where # Iin and # Imax_in represents the actual number of incoming interactions and maximum number of incoming interactions available in a component respectively . The incoming interaction may be defined as a received interface that is required in a component or a received event that arrives at a component. High density shows that a particular component requires so many interfaces.

Metric 4: Component Outgoing Interaction Density (COID)

The COID metric measures the ratio of actual number of outgoing interactions to the maximum number of outgoing interactions available in a component.

$$COID = \frac{\#\ Iout}{\#\ Imax\_out}$$

Where # Iout and # Imax_out represents the actual number of outgoing interactions used and maximum number of outgoing interactions available in a component respectively. The outgoing interaction may be defined as any provided interface used or a source of event consumed.

Metric 5: Component Average Interaction Density (CAID)

The CAID metric is a sum of interaction densities for each component divided by the number of components in software system .

$$CAID = \sum_{i=1}^{i=n} \frac{CIDn}{\#\ components}$$

Where, ∑n CIDn represents the sum of interaction densities for components 1...n and # components represents the number of existing components in the software system.

c) Criticality Metrics

Metric 6: Link Criticality Metric (CRITlink)

The CRITlink metric is defined as the number of components which have links more than a threshold value.

CRITlink = # linkcomponents

Where # linkcomponents represents the number of components, with their links more than a critical value. The threshold is considered as 8 links. The links are created from the facets of other components. If facets increase, criticality of that component increases.

Metric 7: Bridge Criticality Metric (CRITbridge)

The CRITbridge metric is defined as the number of bridge components in a component assembly.

CRITbridge = # bridge_component

Where # bridge_component represents the number of bridge components . A bridge component may be defined as a component which links two or more components/ application. If there is a defect in bridge, the entire application might malfunction. More number of bridge components result in more chances of failure. All the links provided by a bridge component are assigned a similar weight in order to show that they belong to the same bridge component.

Metric 8: Inheritance Criticality Metric (CRITinheritance)

The CRITinheritance metric is defined as the number of components, which become root or base for other inherited components.

CRITinheritance = # root _ component

Where # root_component represents the number of root components which has inheritance. It is the number of components which act as a parent/root/base for other components .

Metric 9: Size Criticality Metric (CRITsize)

The CRITsize metric is defined as below :

CRITsize = # size_component

Where # size_component represents the number of components which exceed a given critical size value. The size is determined by considering the factors like LOC, number of classes, operations and modules in the application. Narasimhan and Hendradjaya defined the threshold value as 1000 lines of code or 50 classes. So, the value for this metric is given as 1 if it exceeds the threshold value.

Metric 10: # Criticality Metric

The #Criticality Metric (CRITall) is defined as the sum of all critical metrics.

CRITall = CRITlink + CRITbridge + CRITinheritance + CRITsize

d) Triangular Metrics

Component Packing Density (CPD) , Component Average Interaction Density (CAID), Component Criticality (CRITall) metrics are considered as 3 axes which can be further modified as 2 axes diagrams with CPD and CAID. For different values varying as high and low for the 2 axes, different cases are considered as the behaviors vary for different systems based on real time, business type etc.

e) Dynamic Metrics

These are the metrics collected during the execution time. These are not available during the design phase as they are collected dynamically. These metrics are used for maintenance purposes.

III. LIMITATIONS OF EXISTING METRICS

The existing metrics have some limitations like most of the existing metrics are applicable to small programs or components, while the objective of having metrics is to test the behaviour and reliability of the components when placed in a large system. Some metrics rely on parameters that could never be measured or are too difficult to measure in practice. Like in case of black box components internal structure may not be available . So there is a need of complexity metric for black box components because a number of existing metrics can not be applied directly. In this paper a metric has been proposed which measures the complexity of a black box component on the basis of component specification.

IV. PROPOSED WORK

In the case of black box components, most of the times source code is not available. So the component consumer has to rely on the component specification to predict its functionality. So the metric proposed in this paper is based on measuring the component complexity on the basis of component specification. This metric uses the different weight values to represent the different complexity levels of components.

The component complexity closely depends on what contributes to develop components, as in [1]. Thus there are four elements that affect the component complexity. First element is Variable Factor that tells complexity of the variables defined in the component. The variables may consist of member variables of a class having scope for the entire class and the parameters, which are local to a particular method. The second element is Interfaces ,which are the access points of component, through which a component can request a service declared in an interface of the service providing component.

Interface complexity is defined as sum of complexity of the interface methods of the class. Third element is Coupling Factor that tells rate of coupling of the methods in the component. Fourth element is cyclometric complexity of the methods of the component.

But in the case of black box components most of the times source code is not available so it is very difficult to guess or find the variables and it is also not possible to find the cyclometric complexity of methods in absence of source code. Thus the metric proposed in this paper includes the concepts of interface methods complexity and coupling complexity between the components ,which can be determined on the basis of component specification. Thus the black box component complexity may be defined as the sum of interface methods complexity and coupling complexity. The CCM(BB) metric has been defined to determine the overall complexity of a black box component.

A. Determining the Interface Method Complexity

In this section a metric for determining the complexity of interface methods has been defined. High interface methods complexity shows more complexity of component.

The interface methods can be divided in the following categories:

- Interface methods having no return value and no parameters.
- Interface methods having return value but no parameters .
- Interface methods having no return value but having parameters.
- Interface methods having return value as well as parameters.

The complexity of the interface methods can be measured on the basis of data types of return value and parameters, and on the basis of number of parameters. On the basis of data type of return value and parameters, and by considering the number of parameters in a method some weights will be assigned to the interface method.

The data types can be divided in the following categories:

- Very simple includes integer,float,double,boolean etc.
- Simple includes structure data types.
- Medium includes class type and object type.
- Complex includes pointer and built in data types.
- Very complex includes user defined data types.

The methods having no return value and no parameters has been considered as simple methods and their weight value has been assumed .025. All other interface methods are assigned weight values depending on the count and data types of parameters, and on the basis of return value's data type. The following Table I represents the weight values assigned to different categories of data types for parameters and return values.

Thus a Interface Method Complexity Metric for Black Box Component, IMCM(BB), has been defined as below:

IMCM(BB) = Wr + PCM(M)

Where Wr represents the weight assigned to the category of return value's data type and PCM(M) is Parameters Complexity Metric for Method which calculate the complexity caused by parameters.

Parameters Complexity Metric for Method ,PCM(M), has been defined as below:

PCM(M) = a*Wvs + b*Ws + c*Wm + d*Wc + e*Wvc

Where a,b,c,d,e represent counts and Wvs,Ws,Wm,Wc,Wvc represent the assigned weights for very simple, simple, medium, complex and very complex data type categories for parameters of a method .

High value of IMCM(BB) shows decrease in understandability and increase in testing effort.

Table I. Represents weight values assigned to different categories of data types for parameters and return values

Parameter Type ,Return Value Type →	Very Simple	Simple	Medium	Complex	Very Complex
Assigned Weight ↓	.10	.20	.30	.40	.50

B. Determining the Coupling Complexity

Component coupling shows the degree of interaction between the components. High coupling results in more component complexity. It will create difficulty in understanding the component behaviour, integrating the component in system, testing the component functionality. A component may interact with other components in order to receive or provide some kind of data. The number of components from which the component receives data(fan-in) and the number of components to which the component provides data(Fan-out), affect the component complexity differently. Thus the component coupling complexity is the sum of coupling complexity caused by fan-in components and fan-out components.

There is one another problem, when a component is coupled with other components then some kind of data is passed between them . But in some cases there may be some problems in exchange of data between them. It will further increase the coupling complexity. Thus it will result in more integration and testing effort.

For example, Suppose return value of one component's method is passed to the another component's method as a parameter to perform its task, but if their data types are different then there will be data type incompatibility problem. So the return value must be converted in the required form before passing as a parameter to second component's method(i.e it needs adaption.). More number of incompatibilities and incompatibilities between more complex data types result in more integration complexity to connect the component with other components to provide accurate functionality. The following Table II shows the assigned weights for complexity in handling the data type incompatibilities, between the different categories of data types.

Thus the coupling complexity metric considers the number of components from which the considered component is receiving data(Fan-in), number of components to which the considered component is providing data (fan out) , number of interactions causing no incompatibility problem, the counts of different types of data type incompatibilities between different data type categories and the weights assigned for handling them.

Thus A Component Coupling Complexity Metric for Black Box Component ,CCCM(BB), has been defined as below:

The number of components from which the considered component receives data(fan-in) and the number of components to which the considered component provides data(Fan-out), affect the component complexity differently. Thus the component coupling complexity may be defined as the sum of coupling complexity caused by fan-in components and fan-out components as shown below:

CCCM(BB) = FICM(BB) + FOCM(BB)

Where FICM(BB) is Fan-in Complexity Metric which measures the coupling complexity due to incoming data from the other components and FOCM(BB) is Fan-out Complexity Metric which measures the coupling complexity due to outgoing data to other components.

Table II. Weights assigned for complexity in handling the data type incompatibilities between the different categories of data types

Actual Data Type → / Converted Data Type ↓	Very Simple	Simple	Medium	Complex	Very Complex
Very Simple	.20	.30	.40	.50	.60
Simple	.30	.40	.50	.60	.70
Medium	.40	.50	.60	.70	.80
Complex	.50	.60	.70	.80	.90
Very Complex	.60	.70	.80	.90	1.0

If the interaction has no data type incompatibility then we have considered the assigned weight as .10 .

Steps to Calculate CCCM(BB)

Step 1 : Calculate FICM(BB)

Fan-in Complexity Metric for Black Box Component, FICM(BB) = fin * [Cn * .10 + (Count the different types of data type incompatibilities need to be handled to receive the data in the correct form and multiply the different counts with their respective weights as shown in Table II and then add them.)]

Where fin is the number of components from which the considered component is receiving data. High value of fin shows that this component's functionality may be affected by many components . Cn represents the count of interactions causing no incompatibility problem.

Step 2: Calculate FOCM(BB)

Fan-out Complexity Metric for Black Box Component, FOCM(BB) = fout * [Cn *.10+ (Count the different types of data type incompatibilities need to be handled to provide the data in the correct form and multiply the different counts with their respective weights as shown in Table II and then add them.)]

Where fout is the number of components to which the considered component is providing data. High value of fout shows that this component may affect the functionality of many components . Cn represents the count of interactions causing no incompatibility problem.

Step 3: Calculate CCCM(BB)

$$CCCM(BB) = FICM(BB) + FOCM(BB)$$

High coupling complexity shows that more integration and testing effort is required. But it represents low maintainability. Because coupling reduces the ease of modification, e.g.,modifying a component affects all the components to which the component is connected.

C. Determine Component Complexity Metric for Black Box Component

Component Complexity Metric for Black Box Component, CCM(BB), has been defined as below

$$CCM(BB) = CCCM(BB) + \sum_{i=1}^{i=n} IMCM(BB)$$

Where n represents the number of methods defined in component interface.

D. Advantages of CCM(BB) Metric

- Easy to understand and use.
- No need of source code, it is based on component specification only.
- Interface Method Complexity Metric provides the estimation of testing effort and understandability. High value of IMCM(BB) for all the interface methods shows more testing effort and less understandability.
- Many coupling metrics consider only the number of interactions to show the extent of coupling. But CCCM(BB) not only considers the number of incoming and outgoing interactions but it also considers the other factors affecting coupling complexity like number of components having impact on the considered component(fin), number of components which may be affected by considered component(fout), number of data items being passed between components and how many of them are creating data type incompatiblity problem. Thus it provides more precise value of component coupling complexity. This metric provides the good indication of component integration and testing effort. High coupling complexity means more integration amd testing effort.
- CCM(BB) includes interface methods complexity and coupling complexity. Thus it gives the overall complexity of component.

V. CONCLUSION

Although the Component Based Software Development is increasingly being adopted for software development. But measuring the black box component complexity during component selection is still a difficult task . Because most of the component complexity measuring metrics are based on source code of the component but in the case of black box components most of the times source code is not provided by the component vendors. So in this paper a component complexity metric has been proposed which is based on component interface specification. By using this metric we can guess the component understandability, testability, integration effort (which is required to integrate this component with other components) and overall component complexity.

Thus by measuring the component complexity during the component selection for component based software development and selecting a less complex component the overall complexity of CBS can be reduced. This will help in reducing the integration and testing effort, and increasing the maintainability.

REFERENCES

[1] Sandeep Khimta, Parvinder S. Sandhu, and Amanpreet Singh Brar, "A Complexity Measure for JavaBean based Software Components", World Academy of Science, Engineering and Technology,2008 .

[2] Luiz Fernando Capretz and Miriam A. M. Capretz, "Component-Based Software Development," The 27th Annual Conference of the IEEE Industrial Electronics Society,2001 .

[3] Ben Whittle and Mark Ratcliffe, "Software Component Interface Description for Reuse," Software Engineering Journal, November 1993.

[4] Usha Kumari and Shuchita Upadhyaya, "An Interface Complexity Measure for Component-based Software Systems," International Journal of Computer Applications , Volume 36– No.1, December 2011.

[5] Chidamber, S. R., Kemerer and C.F, "A Metrics Suite for Object Oriented Design," IEEE Transactions on Software Engineering, 1994,pp. 476-49.

[6] Sedigh Ali, S Gafoor, A. Paul and Raymond A., "Software Engineering Metrics for COTS-based Systems," IEEE Computer, May 2001, pp 44-50.

[7] D. Kafura and S. Henry, "Software Quality Metrics Based on Interconnectivity," Journal of Systems and Software, June 1981, pp 121-131.

[8] Seyyed Mohsen Jamali, "Object Oriented Metrics," Department of Computer Engineering ,Sharif University of Technology, January 2006.

[9] V. L. Narasimhan and B. Hendradjaya, "A New Suite of Metrics for the Integration of Software Components," University of Newcastle , Australia.

[10] Nasib S. Gill and P. S. Grover, "Few important considerations for deriving interface complexity metric for component-based systems," ACM SIGSOFT Software Engineering Notes, March 2004, Volume 29 Issue .

[11] H. Li, "Object-oriented metrics that predict maintainability," Journal of Systems and Software ,Volume 23 Issue 2, 1993, pp: 111-122 .

[12] Dr. P. K. Suri and Neeraj Garg," Software Reuse Metrics: Measuring Component Independence and its applicability in Software Reuse," International Journal of Computer Science and Network Security, VOL.9 No.5, May 2009.

Navneet Kaur received her B.Tech Degree In Computer Science & Engineering from University College Of Engineering, Punjabi University, Patiala, India (2011). Currently she is pursuing her degree for Master of Engineering (ME) In Computer Science & Engineering from Thapar University ,Patiala, India. Her research interests include, Software Components, Software Reuse, Software Architecture and software metrics

Ashima is Assistant Professor in Computer Science and Engineering Department, Thapar University, Patiala, India; and pursuing Ph.D. in Computer Science from Faculty of Engineering, UCOE, Punjabi University, Patiala, India. She holds a Bachelor of Technology (B.Tech.) degree in Computer Science and Engineering from GZSCET, Bathinda, India (2001). She obtained her Master of Technology degree in Computer Science from Department of Computer Science and Engineering, Punjabi University, Patiala India (2005). Her research interests include Software Process Reengineering, Software Engineering, Agile Software Development, Software Quality Improvement in Small Scale Enterprises, Software Reuse, Software Process Customization and Automation, and Software Process Metrics.