DOI: http://dx.doi.org/10.26483/ijarcs.v10i2.6390

ISSN No. 0976-5697

Volume 10, No. 2, March-April 2019

International Journal of Advanced Research in Computer Science

REVIEW ARTICLE

Available Online at www.ijarcs.info

REVIEW OF METRICS EXTRACTION TOOL USING UML

Mala Das
Department of Computer Science,
St. Aloysius' (Auto.) College
Jabalpur (M.P.), India

J. K. Maitra
Department of Mathematics and Computer Science,
Rani Durgawati Vishwa Vidyalaya,
Jabalpur (M.P.), India

Zafar Shareef
Gyan Ganga College of Technology
Jabalpur (M.P.), India

Abstract: In the present consequence, whole world depends on software. It is a cost effective way is essential because all countries depend on complex computer based systems. So it is a big challenge of the developers and researchers to adopt latest technologies which convert a highly complex system design into a simple design. Intended for this purpose, developers inspire the design and construction of computer-based systems by using reusable software which is called as component. A component can be deployed, as they possess the qualities such as reusability, stability, proper communication, modularity, testability, and complexity. The reusable components on integration interoperate with each other resulting in an operational application which is developed with minimum effort and low maintenance cost. We used Component Based Software Development (CBSD) process, which is based on the basic concepts of Object Orientated Techniques where Unified Modelling Language (UML) shows an important role. Different quality factors of a component are measured with the help of metrics, and there are number of metrics proposed for components. In this paper we proposed a methodology of static metrics for integration of software components, complexity metrics for UML Component-based System Specification (CBSS) and interface complexity metrics in a component assembly. These metrics are derived using UML artifacts. We derived these metrics by developing a tool named "CAME" (Component Assembly Metrics Extractor) in NetBeans which parses through the XMI file (XML Meta Data Interchange) generated by UML tool and produces different component metrics through graphic user interface. These metrics will help the software developer in making the system more stable, better and efficient

Keywords: Complexity, CAME, UML, CBSS, CBSD, XML, XMI, CBSE

I. INTRODUCTION

Now these days the software systems are very difficult, bulky and unmanageable. This causes lesser productivity and higher risk management. Software metrics amount different features of software complexity and therefore play an important role in analyzing and improving the quality of software [Diwaker C., et.al., 2014]. The software metrics is used by all people involved in the development processes including customers and managers on one hand, and project leaders, designers and programmers on the other side. As an example, a software manager may be interested in lines of code, while a project manager may be interested in estimating the number of hours required to develop the same number of lines to calculate the productivity of programmers [Abreu F.B., W. Melo., 1996]. Software metric is a mapping from a software development domain to a numerical domain. Here software development domain means an analysis model or the source code of an application and numerical model means the real numbers. A metric must be strongly correlated with a quantitative or qualitative feature of the system. The aims of software metrics are essentially two:

- to give hints on the quality of the system and
- to estimate its development and maintenance costs.

The main purpose of reviewing this paper is to know about, how to design and implement metrics tool? In this paper, one metric tool has designed and implemented named CAME (Component Assembly Metrics Extraction). This tool is used to calculate metrics from UML design documents. It is capable of generating software metrics proposed by researchers for Component Based Software Systems. This paper, demonstrate the CAME tool for a University Case Registration System (UCRS) and its representation in UML and metrics extraction procedure [Pandey; Shareef , 2013].

II. METRICS FOR THE INTEGRATION OF SOFTWARE COMPONENTS

To measure complexity and criticality of large software systems designed and integrated using the principles of CBSE (Component Based Software Engineering), V. L. Narasimhan and B. Hendradjaya, 2004 has proposed two sets of metrics i.e. static and dynamic metrics. Static metrics covered the complexity and the criticality within an integrated component. The static metrics suit includes the CPD metric, CID metric in the entire system. They also define a set of criticality criteria for component integration. By recognizing a complex and/or critical component, it should give a contribution on the effort and cost estimation.

This information should help a software project leader to detect problems at the early stage of the software development. [M. Abdellatief et. al.,2012].

- **COMPLEXITY:** Complexity is an attribute of software. Software is developed based on an algorithm. Complexity of software is dependent on the algorithm used for developing the software. Traditionally, complexity can be defined as the difficulty of analyzing source code, capability of modifying it, and maintaining its different modules.
- **COMPLEXITY METRICES:** For component-based systems, complexity metrics are based on complexity attributes like interaction, coupling, cohesion, interface etc. There is strong demand of complexity metrics for black-box components. The major complexity parameters of black-box components are interface, integration and semantics. An interface act as access points for interaction with the outside computing environment. Integration metrics are the measures of efforts required in the integration process and semantic measures estimate the complexity of relationship of components to an application [Rana and Singh, 2014]. There are several metrics for measuring complexity attributes such as size, control flow, data structures, and inter-module structure. Some of these metrics are –

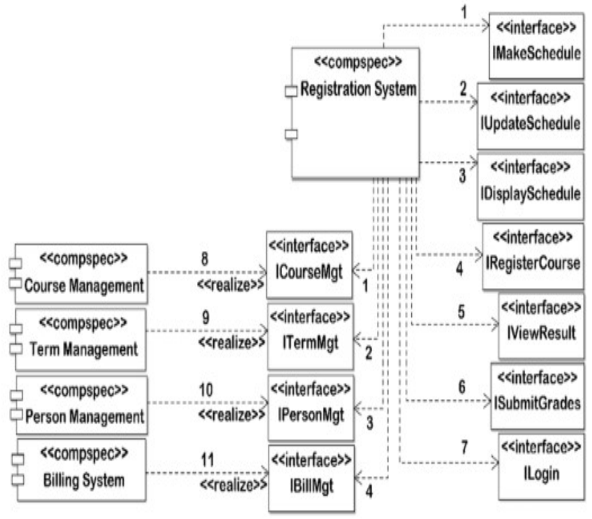
- **METRIC 1 - COMPONENT PACKING DENSITY (CPD)**

The CPD metric is used to identify the density of integrated components. Where, higher density means higher complexity. Here, nature of the component is black box. Where the source code is not available but component developer should provide detailed specification of the component. This specification is used at the early stage of the deriving this metrics.

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

Where, #<Constituent> is the number of lines of code, number of operations for each component, classes. #Components is the number of components in a component assembly. Fig#1 represents a system of integrated components. Each node and link represents a component
Fig#1shows if a student wants to register him/her self in a class. He/she automatically connect with the components of this system such as registration, course, term, person and billing system to perform different operations like to pay fees, to know the class schedule etc.. Here constituent has been taken as number of operations for each component as shown in fig#1. In figure 6 there are 5 components and 24 total operations. The Component Packing Density (CPD) metrics (Metric-1) is derived for a component assembly [Pandey, Shareef, 2013].

and their relationship with other components respectively.



<<compspec>> Registration System
<<interface>> IMakeSchedule
<<interface>> IUpdateSchedule
<<interface>> IDisplaySchedule
<<interface>> IRegisterCourse
<<interface>> IViewResult
<<interface>> ISubmitGrades
<<interface>> ILogin
<<compspec>> Course Management
<<compspec>> Term Management
<<compspec>> Person Management
<<compspec>> Billing System
<<interface>> ICourseMgt
<<interface>> ITermMgt
<<interface>> IPersonMgt
<<interface>> IBillMgt
<<realize>>
1
2
3
4
5
6
7
8
9
10
11

Fig#1: Components and Interfaces in a Component Assembly Model [Mahmoodand Lai, 2006].

$$\text{CPD}_{\text{Operation}} = \frac{\text{Number of Operations}}{\text{Number of Components}}$$

$$\text{CPD}_{\text{Operation}} = 24/\ 5 = 4.8$$

The result for $\text{CPD}_{\text{Operation}}$ which is 4.8 is calculated by the CAME tool. [Pandey, Shareef, 2013].

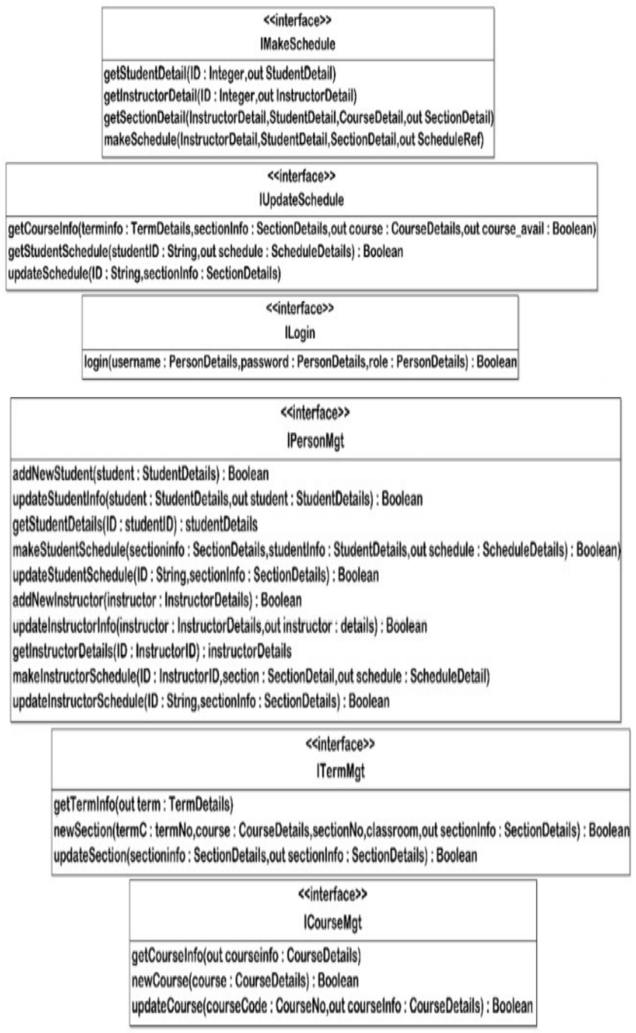
SIGNIFICANCE: For CPD metrics the number of operations and number of components in a component assembly are taken into account. As per model UCRS, when the number of components increases, density increases. A higher density means a higher complexity, which will force the developer to spend more effort and risk assessment on the system. Therefore more resources are needed to complete the project.

- **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. When a component provides an interface and other components use it and also when a component submits an event and other components receive it, then it is called an interaction. It can be derived as the interfaces of component are provided in UML diagram as fig#1 and 2.

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

Where, #I represents the number of actual interactions. #Imax represents the number of maximum available interactions. The component “Registration System” is having 4 actual interactions, which are required interfaces and 7 are provided interfaces which are not being used, thus total interactions in a component are 11.


<<interface>>
IMakeSchedule
getStudentDetail(ID : Integer,out StudentDetail)
getInstructorDetail(ID : Integer,out InstructorDetail)
getSectionDetail(InstructorDetail,StudentDetail,CourseDetail,out SectionDetail)
makeSchedule(InstructorDetail,StudentDetail,SectionDetail,out ScheduleRef)
<<interface>>
IUpdateSchedule
getCourseInfo(terminfo : TermDetails,sectionInfo : SectionDetails,out course : CourseDetails,out course_avail : Boolean)
getStudentSchedule(studentID : String,out schedule : ScheduleDetails) : Boolean
updateSchedule(ID : String,sectionInfo : SectionDetails)
<<interface>>
ILogin
login(username : PersonDetails,password : PersonDetails,role : PersonDetails) : Boolean
<<interface>>
IPersonMgt
addNewStudent(student : StudentDetails) : Boolean
updateStudentInfo(student : StudentDetails,out student : StudentDetails) : Boolean
getStudentDetails(ID : studentID) : studentDetails
makeStudentSchedule(sectioninfo : SectionDetails,studentInfo : StudentDetails,out schedule : ScheduleDetails) : Boolean)
updateStudentSchedule(ID : String,sectionInfo : SectionDetails) : Boolean
addNewInstructor(instructor : InstructorDetails) : Boolean
updateInstructorInfo(instructor : InstructorDetails,out instructor : details) : Boolean
getInstructorDetails(ID : InstructorID) : instructorDetails
makeInstructorSchedule(ID : InstructorID,section : SectionDetail,out schedule : ScheduleDetail)
updateInstructorSchedule(ID : String,sectionInfo : SectionDetails) : Boolean
<<interface>>
ITermMgt
getTermInfo(out term : TermDetails)
newSection(termC : termNo,course : CourseDetails,sectionNo,classroom,out sectionInfo : SectionDetails) : Boolean
updateSection(sectioninfo : SectionDetails,out sectionInfo : SectionDetails) : Boolean
<<interface>>
ICourseMgt
getCourseInfo(out courseinfo : CourseDetails)
newCourse(course : CourseDetails) : Boolean
updateCourse(courseCode : CourseNo,out courseInfo : CourseDetails) : Boolean

Fig#2: Component Assembly and Interfaces of UCRS Model [Mahmoodand Lai, 2006].

$$\text{CID Registration-system} = \frac{\text{Number of Actual Interaction}}{\text{Number of total Interaction}}$$

CID Registration-system = 7/11, which is 0.3636

The result for CID Registration-system which is 0.3636 is calculated by the CAME tool [Pandey, Shareef, 2013].

SIGNIFICANCE: For CID metrics the number of Actual Interaction and Number of total Interaction in a component assembly are taken into account. As per model UCRS, there is a risk in submitting and receiving an event, as events have to be handled with care for correct processing. The main problem arises when measuring the density of interactions in a component. When the density of interaction increases, complexity increases.

- **METRIC 3 - COMPONENT INCOMING INTERACTION DENSITY (CIID)**

The CIID metric measures the ratio of actual number of incoming interactions to the maximum available incoming interactions in a component, which can be obtained from design document. Incoming interaction is defined as a received interface that is required in a component or a received event that arrives at a component.

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

Where, # Iin represents the actual number of incoming interactions; # Imax_in represents the maximum number of incoming interactions available in a component. The component “Registration System” has 4 actual incoming interactions and total of incoming interactions for this component is also 4.
CIID Registration-System

$$= \frac{\text{Number of actual incoming interactions}}{\text{Number of total incoming interactions}}$$

CIID Registration-system = 4/4, which is 1.0 , The result for CIID Registration-system which is 1.0 is calculated by the CAME tool [Pandey, Shareef, 2013].

SIGNIFICANCE: As per model UCRS, High density shows that a particular component requires so many interfaces. A higher density of CIID shows that a particular component needs extra effort to examine all received interfaces or events.

- **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. Which is for outgoing interaction to available outgoing interactions can be obtained at early stage.

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

Where, # Iout represents the actual number of outgoing interactions used; # Imax_out represents the maximum number of outgoing interactions available in a component. This component “Registration System” has 7 total available provided interfaces, but is not being used by any component, so actual provided interfaces, which are being consumed by other components are 0.

COID Registration-System

$$= \frac{\text{Number of actual outgoing interactions}}{\text{Number of actual outgoing interactions}}$$

COID Registration-system = 0/7, which is 0.

The result for COID Registration-system which is 1.0 is calculated by the CAME tool [Pandey, Shareef, 2013].

SIGNIFICANCE:. As per model UCRS, Outgoing interactions are any provided interface used and any possible source of events consumed. This metric calculates density in a component. A higher density of COID shows that a particular component needs extra effort to examine all provided interfaces or send events.

- **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 is calculated based on previous values can also be obtained from design documents.

$$\text{CAID} = \sum_{i=1}^{n} \frac{\text{CID}_n}{\#\text{ Components}}$$

Where, ∑ CID_n represents the sum of interaction densities for components 1...n; # components represents the number of existing components in the software system. There are 5 components, which have "provided" and "required" interfaces. The components "Course Management", "Term Management", "Person Management", "Billing System" provided interfaces are being consumed, where as "Registration System" is having 7 provided interfaces, but they are not being consumed.

(i) $\text{CID}_{\text{Registration System}}$	=	4/11	= 0.3636
(ii) $\text{CID}_{\text{Course Management}}$	=	1/1	= 1
(iii) $\text{CID}_{\text{Term Management}}$	=	1/1	= 1
(iv) $\text{CID}_{\text{Person Management}}$	=	1/1	= 1
(v) $\text{CID}_{\text{Billing System}}$	=	1/1	= 1

$$\text{CAID} = \frac{\sum \text{CID}_n}{\text{Number of Components}}$$

CAID =

$$\frac{\text{CID}_{\text{RegistrationSystem}}+\text{CID}_{\text{CourseManagement}}+\text{CID}_{\text{TermManagement}}+\text{CID}_{\text{PersonManagement}}+\text{CID}_{\text{Billing System}}}{\text{Number of Components}}$$

CAID = (0.3636+1+1+1+1)/5 = 0.872723
CAID = 4.3636/5 = 0.872723
The result for CAID which is 0.872723 is calculated by the CAME tool [Pandey, Shareef, 2013].

SIGNIFICANCE: As per model UCRS, It evaluates the complexity of the entire component assembly. The value will be valuable to assess the whole system interaction. The low value of CAID indicates low interactions, which also means lower complexity.

III. CRITICALITY METRICS

Criticality metrics used for a critical component that binds a system, consisting of assembly of components. For a software tester, this component requires substantial testing effort. Every possible scenario for this critical component has to be tested, particularly if it is a base component, so that any incorrect operations are not inherited by the sub-components. To identify the critical components, or the circumstances that make a component critical, four metrics are used and they characterize the circumstances that make a component critical. These metrics are Link Criticality, Bridge Criticality, Inheritance Criticality and Size Criticality metrics.

- **METRIC 6 - LINK CRITICALITY METRIC ($\text{CRIT}_{\text{LINK}}$)**

Link Criticality metric is defined as the number of components which have links more than a threshold value. It specified a component to be called critical if the number of links has reached a certain threshold value. In UML these links can be easily represented, hence the metric can be obtained at early stage. Links are created from the provided interfaces of other components.

$\text{CRIT}_{\text{link}}$ = # linkcomponents

Where, # linkcomponents represents the number of components, with their links more than a critical value. The threshold is considered as 8 links. This metric can be empirically understood with the help of a model as shown in the fig#1 and 2. The component "Term Management" has only one provided interface, so link (Term-management) or $\text{CRIT}_{\text{link}}$ for this component is displayed as "Not Critical". If a component link exceeds a threshold value, then the links of that component will be displayed as "Critical".

SIGNIFICANCE: As per model UCRS, A component will be called critical, if the number of links has reached a certain threshold value. At this stage, it does not have the exact threshold value. If the number of provided interfaces increases, criticality of that component increases.

- **METRIC 7: BRIDGE CRITICALITY METRIC ($\text{CRIT}_{\text{BRIDGE}}$)**

It is used for detecting components acting as bridge between two components in a component assembly. It is derived by counting the number of components acting as bridge (link) between two components as proposed by Narasimhan and Hendradjaya [Narasimhan and Hendradjaya*,* 2007], hence it can be derived at early stage.

$\text{CRIT}_{\text{bridge}}$ = # 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 whole application might malfunction. More number of bridge components means more chances of failure. There are no components acting as bridge, because all the four components "Course Management", "Term

Management", "Person Management", "Billing System" are providing interfaces directly to component "Registration System". So, the component "Registration System" for $CRIT_{bridge}$ result is zero.

SIGNIFICANCE: As per model UCRS, to identify a bridge component, an importance weight should be placed for each link by the developer based on their experience. This component has to be identified, since a defective bridge component has a high probability to prevent the functioning of the entire application. The more the number of bridge components implies, the more the chances for failure.

- **METRIC 8 - INHERITANCE CRITICALITY METRIC ($CRIT_{INHERITANCE}$)**

Inheritance Criticality metric is defined as the number of components, which become root or base for other inherited components. This metric can be obtained at early stage because those components are identified which becomes root or base for other inherited components.

$$CRIT_{inheritance} = \# root\_component$$

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.
All the four components "Course Management", "Term Management", "Person Management", "Billing System" are providing interfaces directly to component "Registration System", which also has provided interfaces, which are not being consumed. These components are not exhibiting hierarchical form. For Inheritance Criticality metric a component to be acting as root, base or parent component where other components are linked to each other in hierarchical order. Since there is no component acting as root component, so the result for the component "Registration System" is zero.

SIGNIFICANCE: For $CRIT_{inheritance}$ metrics the number of root components which has inheritance in a component assembly is taken into account. As per model UCRS, The base component introduces the risk of constructing the right information to be inherited. The more the number of base components, the higher is the potential associated risks.

- **METRIC 9 - SIZE CRITICALITY METRIC ($CRIT_{SIZE}$)**

Size criticality metrics ($CRIT_{size}$), determines the size of the component, and the component becomes critical if it exceeds the threshold value. The size of component at early stage can be easily obtained from information that comes with the associated detailed specification. Size Criticality metric is defined as below:

$$\mathbf{CRIT_{size} = \# size\_component}$$

Where # size_component represents the number of components which exceed a given critical size value. The size is defined in terms of LOC, number of classes, operations and modules in the application. Narasimhan and Hendradjaya [Narasimhan and Hendradjaya, 2007] defined the threshold value as 1000 lines of code or 50 classes. The component model proposed in this section consists of components being black box in nature; only their interfaces can be accessed, which contains operations. These, are counted to check the threshold value, so the value for this metric is given as 1 if it exceeds the threshold value for a particular component.
This metric can be empirically understood with the help of a model as shown in the figure 6 and 7. The Size Criticality Metric helps in detecting components whose size value exceeds a threshold value, which is defined by Narasimhan and Hendradjaya, components being black box in nature, only their interfaces can be accessed, which contain operations. These are counted to check the threshold value. So, the value of this metric is 1 if it exceeds the threshold value for a particular Component. If it exceeds the value then that particular component is considered "critical" otherwise it is "simple". If the component does not contains any operations than it is displayed as "Not Available". Here the interface "ITermMgt" of component "Term Management" is having maximum operations i.e. 3, so it does not exceed threshold value and is considered "simple".

- **METRIC 10 - # CRITICALITY METRIC**

The #Criticality Metric ($CRIT_{all}$) is defined as the sum of all critical metrics.

$$\mathbf{CRIT_{all} = CRIT_{link} + CRIT_{bridge} + CRIT_{inheritance} + CRIT_{size}}$$

By totalling the number of components that have link, bridge, inheritance and size criticality, we obtain the criticality level of the component assembly. The value of CRIT is compared to a threshold value in order to identify the criticality level of a component assembly. For instance, if the threshold value for $CRIT_{link}$, $CRIT_{bridge}$, $CRIT_{inheritance}$, and $CRIT_{size}$ equal 5, then the threshold value for $CRIT_{all}$ is 20. By experimenting with more empirical data, a more accurate threshold value could be produced.

IV. METHODOLOGY

The objective of this paper is an assembly of components for a system to be designed at early stage. In this research work, by using open source UML "ArgoUML", a tool using NetBeans has been developed, named as "CAME" (Component Assembly Metrics Extractor), this tool helps in extracting the static complexity metrics proposed by Narasimhan and Hendradjaya [Narasimhan and Hendradjaya, 2007]. Using the ArgoUML tool, the model given in is designed creating component artifacts through Deployment diagram option, the XMI 1.2 file is generated with the help of Export XMI option (ArgoUML using Netbeans XMI Writer version 1.0), which is then parsed for extracting information related to various metrics in a component assembly for component-based systems using a Java based software tool. The parser parses the XMI file, which contains information about all the components integrated into the system, this open source tool assigns a unique XMI identifier (UUID) to flag user designed model components. For example, in 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, or 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. This model provides an overall view of the system and helps to demonstrate the extraction of existing component assembly complexity metrics.

With the help of CAME tool a number of facts related to component assembly can be derived through XMI file, which helps the developer in analyzing the different aspects of components and assembly information. This information is displayed through User Interface. Through this user interface a developer can learn about number of components, their interfaces and the operations available in an interface of a component assembly. Component names are displayed and after selecting a particular component its interfaces: Provided (Abstraction) and Required (Dependency) can be displayed. Similarly the details of operations of a particular interface can be displayed by selecting the interface. The component-"Registration System" has a provided interface "IMakeSchedule", this interface consists of four operations, the same information is provided using CAME tool.

In this work static metrics are used for integration of software components, complexity metrics for UML Component-based System Specification (CBSS) and interface complexity metrics in a component assembly proposed by different authors for component-based systems using UML artifacts are derived. These metrics are derived by developing a tool named "CAME" (Component Assembly Metrics Extractor) in Netbeans which parses through the XMI file (XML Meta Data Interchange) generated by UML tool and produces different component metrics through graphic user interface. These metrics will help the software developer in attaining more information at early stage, making the system more stable, better and efficient.

V. COMPONENT ASSEMBLY METRICS EXTRACTOR (CAME) TOOL

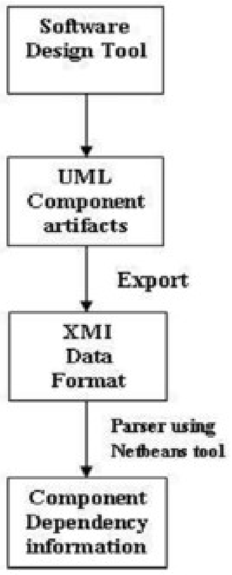
To make use of component assembly metrics and interface metrics suite, the CAME tool (Component Assembly Metrics Extractor) is used. It is a java-based tool developed in NetBeans to analyze UML Component Assembly diagrams represented in XML-based formats, namely XMI. It is extract existing metrics for component assemblies using XMI files for Component-based systems. This tool is limited to component diagram, collaboration diagram and interfaces only so it works only with those XMI files that contain these features. A user draws UML component diagrams or collaboration diagrams with only the elements provided by the ArgoUML tool.

XMI input derived from UML models created by software design tools is transformed as follows [Varol, 2005] –

• UML model to XMI file generated

• XMI input to parser

• Different status of Components related to dependency output using Netbeans software tool.



Fig#3: XMI input derived from UML models created by software design tools is transformed.

- **EXTENSIBILITY:** CAME TOOL PROVIDES A USER INTERFACE WHERE A USER CAN SELECT A UML COMPONENT ASSEMBLY MODEL IN ARGOUML AND CAN COMPUTE DIFFERENT COMPONENT ASSEMBLY METRICS.

- **DATA EXPORT:** CAME TOOL COMPUTE METRICS FOR COMPONENT ASSEMBLY THROUGH XMI FILE AND DISPLAYS THEM THROUGH USER INTERFACE.

- **MULTI-PLATFORM SUPPORT:** CAME TOOL WILL RUN ON ALL PLATFORMS THAT SUPPORT THE JAVA 1.2 OR HIGHER RUNTIME ENVIRONMENT (WINDOWS 9X/ME/NT/2000/XP/WINDOWS 7, UNIX AND LINUX).

VI. SOFTWARE METRICS EXTRACTION USING UML

The main focus of this review is mainly on how to obtain more information from UML artifacts such as component and collaboration diagram, through XMI information sheet, errors which occur at early stage can be corrected and can be stopped from migrating to later stages. To collect this useful information CAME tool is used. It extracts information regarding the component assemblies characterized through metrics defined by Narasimhan and Hendradjaya [2007] Mahmood and Lai [Mahmood and Lai, 2006] through XMI information sheet.

VII. CONCLUSION

This review work concludes that the component based software engineering is the efficient approach for dealing with the higher complex and real time software systems. The metrics proposed by the Narasimhan and Hendradjaya

[2007] are specifically proposed for design documents like UML class diagrams; these diagrams representing the static nature, however this paper does not provide any empirical validation of these metrics, but proposes the validation for future work. The proposed metrics [Narasimhan and Hendradjaya, 2007] have been extracted from design documents using CAME tool [Pande; Shareef, 2013] where only static metrics for component assembly are extracted.

VIII. FUTURE RESEARCH

In this paper, CAME tool extracted the metrics from design documents (UML) using component diagram and their interfaces but here this work is limited to static metric only. The developed tool works for only component models developed in ArgoUML, this can be further upgraded for other UML tools like Rational Rose, Magic Draw UML, UMLet, ESS-Model. The tool can be further upgraded for extraction of dynamic metrics for component-based systems. Other metrics related to Component-based systems can be included in enhanced version of the tool proposed.

REFERENCES

[1] Abreu F.B., Melo W., (1996). Evaluation the Impact of Object-Oriented Design on Software Quality, Proceedings of the 3rd International Software Metrics Symposium, Berlin, Gennany, pp. 90-99, March.

[2] Abdellatief M. et. al., (2012). Multidimensional Size Measure for Design of Component-Based Software System, Institute of Software and Technology, Vol. 6, pp. 350–357.

[3] B. Boehm, et al., (2000). Software Cost Estimation with COCOMO II. Prentice Hall.

[4] Bakshi A., Singh R., (2013). Component Based Development in Software Engineering, International Journal of Recent Technology and Engineering (IJRTE), ISSN: 2277-3878, Volume-2, Issue-1, March, pp. 1, 48-52.

[5] Bayar V., (2001). A Process Model for Component Oriented Software Development, Master Thesis.

[6] Booch, G., (1994). Object-Oriented Analysis and Design with Applications, 2nd ed., Benjamin Cummings.

[7] Bellin, D., Tyagi M., et al., Object-Oriented Metrics:An Overview , Computer Science Department,North Carolina A ,T state University,Greensboro,Nc 27411-0002.

[8] Beshar Dhaya Nor, (2015). Comparative Analysis Of Software Reusability Attributes In Web And Mobile Applications, University Tun Hussein Onn Malaysia, April.

[9] Capers J., (2012). A Short History Of The Lines Of Code (Loc) Metric ,Version 6.0. December 2

[10] Churcher, N. I. and Shepperd, M. J.,(1995). Comments on 'A Metrics Suite for Object-Oriented Design', IEEE Transactions on Software Engineering, vol. 21, pp. 263-5.

[11] Clemens S., (1998). Component Software: Beyond Object-Oriented Programming, Addison Wesley.

[12] Chawla S., Kaur G.,(2013). Comparative Study Of The Software Metrics For The Complexity And Maintainability Of Software Development, Journal Of Advanced Computer Science and Application (IJACSA) Vol. 4, No. 9.