Actual Interactions for Component-Based Software using Straight and Circuitous Links

Umesh Kumar Tiwari, Santosh Kumar

Abstract*: Interaction and integration complexities of various pieces of codes play a vital role in the overall behavior of software. As the code count increases the interaction level of software also increases as per the requirements of the software. In this paper we propose a metric to assess the actual number of interactions made by components in component-based software. On the basis of interactions among components we define an Interaction-graph. Interaction-graph contains 'Links' and 'Components'. To assess the actual interactions we define inner and outer interactions of particular components. Links are further categorized as straight and circuitous links. Proposed interaction metric is easy to calculate and contains information about the component which is used by designers and developers of the component-based software for future development.

***Index Terms*: Component-based software, metric, Interaction-graph, Links, Inner-Interactions, Outer-Interactions.**

I. INTRODUCTION

In general, complexity is termed as the assessment of hardware and software resources needed by software. In software development, complexity is treated as an indirect measurement unlike the direct measurements like lines-of-code or cost-estimation [1]. Internal as well external interactions contribute a major role in software complexity. In the context of software development, interaction behaviour of various parts of program is used to measure the complexity. These parts may be single line code, a group of line of codes (functions), a group of functions (modules) or ultimately components. As the size of parts of s software increases, the count of interactions will also increase, as well as the complexity.

Software Engineering principles are applicable on the applications developed through either development paradigm. Component-based software development (CBSD) emphasizes "development with reuse" as well as "development for reuse". Development with reuse focuses on the identification, selection and composition of reusable components. The property of reusability is not applied only to develop the whole system but also to develop the individual components. The development for reuse is concerned with the development of such components that may be used and then reused in many applications, in similar and heterogeneous contexts.

After discussing the introduction of work in section 1, we have summarized the interaction and integration issues in section 2. In section 3, we have performed the survey on the literature available. Section 4 includes the proposed work. It also includes an exemplar case study to implement the proposed work. Finally section 5 concludes this work.

II. INTEGRATION AND INTERACTION ISSUES

Software applications are composed of dependent or independently deployable components. Assembling of these components has a common intension to contribute their functionalities to the system. Technically this assembling is referred to as integration of and interaction among components. We have sufficient number of measures and metrics to assess the complexity of stand alone programs as well as small-sized conventional software, suggested and practiced by numerous practitioners [2]-[8]. In literature, complexity of programs and software is treated as a "multidimensional construct" [3], [9].

III. LITERATURE SURVEY

A. Thomas J. McCabe's Cyclomatic Complexity

Thomas J. McCabe [10] developed a method to assess the Cyclomatic complexity of a program. He used control-flow graph of code to compute the complexity. McCabe used graph theoretic notations to draw the control-flow graph where a graph denoted as 'G' having 'n' number of nodes, 'e' number of connecting edges and 'p' number of components. Cyclomatic complexity V(G) calculated as, V(G) = e - n + 2p, where 2 is the "result of adding an extra edge from the exit node to the entry node of each component module graph" [2]. In control-flow graph, a sequential block of code or a single statement is represented as a node, and control flows among these nodes are represented as edges. Cyclomatic complexity metric is easy to compute and maintenance, gives relative complexity of various designs.

Method:

McCabe used a set of programs developed in FORTRAN language to illustrate his implementations. McCabe used graph theoretic notations to draw the control-flow graph where a graph denoted as 'G' having '*n*' number of nodes, '*e*' number of connecting edges and '*p*' number of components.

Cyclomatic complexity V(G) calculated as,

$$V(G) = e - n + 2p,$$

where, 2 is the "result of adding an extra edge from the exit node to the entry node of each component module graph" [2]. In a structured program where we have predicate nodes, complexity is defined as,

Revised Manuscript Received on January 05, 2020

Umesh Kumar Tiwari, Computer Science and Engineering, Graphic Era Deemed to be University, Dehradun, India.

Santosh Kumar, Computer Science and Engineering, Graphic Era Deemed to be University, Dehradun, India.

$$V(G) = \text{number of predicate nodes} + 1$$

Where predicate nodes are the nodes having two and only two outgoing edges.

In his implementations, McCabe has defined the value of Cyclomatic complexity of a program as less than 10 as reasonable. If a program has hierarchical structure, that is, one subprogram is calling other one, the Cyclomatic complexity is the summation of individual complexities of these two subprograms and is given as,

$$V(G) = v(P_1 + P_2) = v(P_1) + v(P_2),$$

where, P_1 and P_2 are two subprograms and P_1 is calling P_2.

Key Findings:

- Complexity does not depend on the size, but the coding structure of the program.
- If a program has only one statement then it has complexity 1. That is, $V(G) \geq 1$.
- Cyclomatic complexity V(G) actually defines the number of independent logics/paths in the program.

Metrics Used:

- Lines of code,
- Control flow of statements,
- Interaction among statements,
- Independent paths from source to destination,
- Vertices and edges.

Factors affecting Interaction and Integration Complexity:

- Structure of the program,
- Forward and backward loops,
- Branching statements,
- Switch cases in the program.

Critique:

- Same program written in different languages or with different coding style or structure may have different complexities.
- Intra-module complexity of simple structured programs can be achieved easily, but for inter-module complexity, this metric produces misleading output.

B. Halstead's Software Science

Halstead's [5] identified a complete set of metrics to measure the complexity of a program considering various factors. These metrics include the program vocabulary, length, volume, potential volume, and program level. Halstead proposed methods to compute the total time and effort to develop the software. These metrics are based on the lines of codes of the program. He defined program vocabulary as the count of distinct operators and distinct operands used in the program. The count of total operators and operands used in a program is proposed as the Program length. The Program volume has been defined as the storage volume required representing the Program, and the representation of program in the shortest way without repeating operators and operands is known as potential volume. Halstead has also defined the relationship between these factors and metrics of programs.

Method:

Halstead proposed software science to examine the algorithms developed in ALGOL and FORTRAN. Halstead considered the algorithms/programs as a collection of 'tokens', that is, operators and operands. He defined program vocabulary as the count of distinct operators and distinct operands used in the program. The count of total operators and operands used in a program is proposed as the Program length. The Program volume has been defined as the storage volume required representing the Program, and the representation of program in the shortest way without repeating operators and operands is known as potential volume.

$$\text{Program vocabulary: } n = n_1 + n_2,$$

where n_1 and n_2 are the count of unique operators and operands respectively,

$$\text{Program length } N = N_1 + N_2,$$

where N_1 and N_2 are the count of total operators and operands respectively.

They further proposed if the program is assumed to contain binary encoding then the size is defined as program volume and can be defined as-

$$\text{Program volume } V = N \times \log_2 n = (N_1 + N_2) \times \log_2 (n_1 + n_2),$$

where $\log_2 n$ is used for binary search method.

An algorithm can be implemented in various efficient and compact ways. They defined the most competent and compact length of the program as potential volume. For a program potential volume can be attained by specifying signature (name and parameters) of functions and subprograms previously defined and formulated as-

$$\text{Potential volume } V^* = (2 + n_2^*) \times \log_2 (2 + n_2^*),$$

where, 2 represents the two operators (one for name of the function and other the separator used to distinguish the number of parameters) and n_2^* represents the operand used for the count of input and output parameters.

Next he defined the level of a program where level is the possible minimum size of the program. A program having volume V and potential volume V*, the program level is defined as-

$$\text{Program Level (L) } L = V^*/V$$

Where $0 \leq L \leq 1$, 0 denotes the maximum possible size and 1 denotes the minimum possible size of the program.

On the basis of level of program, Halstead defined the difficulty of writing a program as-

$$D = 1/L$$

Where, difficulty is the inverse of the program level.

Further he defined the effort metric to develop a program as-

$$E = V/L = D \times V$$

As the volume and difficulty of program increases, the effort of development increases.

Key Findings:

- A range of complex metrics and their values are achieved using simple measures including operators, operands and size of the algorithm.
- There is no in-depth analysis requirement of structure of the logic code; hence the ease of computation makes proposed metrics achievable and can be comfortably automated.

Metrics Used:

- Operators and Operands,
- Functions and subprograms,
- Input/Output parameters.

Factors affecting Interaction and Integration Complexity:

Count of operators, operands, function names and similar measures.

Critique:

- Originally software science was proposed to investigate the complexity of algorithms not the programs, therefore these metrics are static measures.
- Halstead tested their metrics on small scale programs even less than 50 statements. So applicability on large programs is questionable. These small scale metrics cannot be generalize with respect to large, multi-module programs/software.

In his theory Halstead calculated each occurrence of GOTO statement as a distinct operator whereas he treated all the occurrences of an IF statement as single operator. Treating and counting different operators as different may create ambiguity.

C. Alan Albrecht's Function Point Analysis

Alan Albrecht [6] proposed Function-point analysis technique to measure the size of a system in terms of functionalities provided by the system. FPA categorizes all the functionalities provided by the software in five specific functional units: External inputs provided to the software, External outputs provided by the software, External inquiries of the system under consideration, Internal logical files presents data and content residing in the system, and External interface files are the data and contents residing with other systems and can be called to system under consideration. Three complexity weights High, Low and Medium are associated with these functional units using a set of pre-defined values. In function-point analysis, 14 complexity factors have been defined, which have a rating from 0 to 5. On the basis of these factors, Alan calculated the values of unadjusted function-point, complexity adjustment factors, and finally the value of function points [2].

Method:

FPA categorizes all the functionalities provided by the software in five specific functional units:

External inputs are the number of distinct data inputs provided to the software or the control information inputs that modifies the data in internal logical files. Same inputs provided with the same logic are not included in the count for every occurrence. All the repeated formats are treated as one count.

External outputs are the number of distinct data or control outputs that are provided by the software. Same outputs achieved with the same logic are not included in the count for every occurrence. All the repeated formats are treated as one count.

External inquiries are the number of inputs or outputs provided to or achieved from the system under consideration without making any change in the internal logical files. Same inputs/outputs with the same logic are not included in the count for every occurrence. All the repeated formats are treated as one count.

Internal logical files presents the number of user data and content residing in the system or control information produced or used in the application.

External interface files are the number of communal data, contents, files or control information that is accessed, provided or shared among the various applications of the system.

These five functional units are categorized into three levels of complexity: low/simple, average/medium, or high/complex. Albrecht identified and defined weights for these complexities with respect to all the five functional units. Now these functional units and corresponding weights are used to count the unadjusted function points, as-

$$\text{Unadjusted FP} = \sum_{i=1}^{5}\sum_{=1}^{3}(\text{Count of Functional Unit} * \text{Weight of the Unit})$$

Where, 'i' denotes the five functional units and 'j' denotes the level of complexity.

Similarly, Albrecht defined the Complexity adjustment factors on the basis of 14 complexity factors on a scale of 0 to 5. Adjustment factors provide an adjustment of +/- 35% ranging from 0.65 to 1.35. These complexity factors include-reliable backup and recovery, requirement of communication, distributed processing, critical performance, operational environment, online data entry, multiple screen inputs, updation of master files, complex functional units, complex internal processing, reused code, conversions, distributed installations, and ease of use. Complexity factors are rated as-no influence (0), Incidental (1), Moderate (2), Average (3), Significant (4), and Essential (5).

Complexity adjustment factor is defined as-

$$\left[0.65 + 0.01 * \sum_{k=1}^{14}(\text{Complexity Factor})_i\right]$$

Now the function point is defined as the product of Unadjusted FP and Complexity adjustment factor.

$$\text{FP} = \text{Unudjudted FP} * \text{Complexity adjustment factor}$$

Key Findings:

- Function point technique does not depend on tools, technologies or languages used to develop the program or software. Two dissimilar programs having different lines of code may provide same number of function points.
- These estimations are not based on lines of code hence estimations can be made early in the development phase, even after the commencement of the requirements phase.

Metrics Used:

- Count of inputs, outputs, internal logical files, external interfaces and enquiries.
- Weights of corresponding functional unit on the scale of low, medium and high.
- 14 complexity factors on the rating of values 0 to 5.

Factors affecting Interaction and Integration Complexity:

Count of functions in the software.

Critique:

- To compute correct count of function points, proper analysis of requirements by trained analysts is required.
- Analysis, counts of functional units and computation of function points are not as simple as counting of lines of code.

D. Henry and Kafura's Complexity Metric

Henry and Kafura [11] proposed a set of complexity computation method for software modules. Author's suggested a "Software Structure Metrics Based on Information Flow that measures complexity as a function of fan-in and fan-out" [12]. Authors proposed the complexity as "the procedure length multiplied by the square of fan-in multiplied by fan-out." This method is used to calculate the count of "local information flows" coming to (fan-in) and going from (fan-out) the module. Henry and Kafura defined a length of the module as the procedure length which calculated with the help of LOC or McCabe's complexity metric. This metric can be computed comparatively early stage of the development.

Method:

Henry and Kafura defined three categories of data flow in their work:

Global flow- It is defined when a global data structure is involved between two modules. One module submits its data to the global data structure and the other module accesses that submitted data from the data structure.

Direct local flow- Flow of data between two modules is direct local if one module directly calls another module.

Indirect local flow- Flow of data between two modules is indirect if one module uses data as an input returned by some other module or both these modules were called by some third module.

Complexity metrics are defined on the basis of two types of information flow for a particular module or procedure-

Fan-In- It defines the sum of number of local flows coming to the module and the count of data structures used to access the information.

Fan-Out- It defines the sum of number of local flows going from the module and the count of data structures modified by the module.

Authors proposed the local flow complexity as "the procedure length multiplied by the square of fan-in multiplied by fan-out." This method is used to calculate the count of "local information flows" coming to (fan-in) and going from (fan-out) the module. That is-

Complexity in terms of local flows = length of the module (fan-in flows of the module * fan-out flows of the module) ^{2}

High values of fan-in and fan-out indicates the high coupling among modules which leads the problem of maintainability.

Global flow complexity is defined in terms of possible read, write and read-write operations made by the procedures of the module. That is-

Global information in terms of access and update = (write * read) + (write * read-write) + (read-write * read) + (read-write * (read-write - 1))

Key Findings:

- The type, nature, number, format of the information which is going to transit among the software components are identified and defined much before the actual implementation. Therefore these metrics can be applied and estimated at the time of design phase.
- These design phase metrics can be used to identify the shortcomings and flaws in the construction of design of procedures and ultimately of modules.
- Through their metrics authors argued that the size of the code plays negligible role in complexity estimation.

Metrics Used:

- Data and information transit among modules.
- Number of parameters used to access and to provide information.

Factors affecting Interaction and Integration Complexity:

- Number of incoming and outgoing flows
- Number of parameters used to access and modify data structure
- Number of operations updating the data structure.

Critique:

- Author's computed length with the help of McCabe's formula or Halstead's formula, that is, length of the code plays a vital role in the metrics.
- If the module has no interaction with other modules then the complexity of that module becomes zero.

In global information flow, only update operations are participating in the complexity.

E. Cho et al. 's Complexity Metric

Cho et al. [13] developed some measures to quantify the quality and complexity of CBSE components. In their work, authors defined three categories of complexity measures: complexity, customizability and reusability of a component. Some of these measures are applicable to design phase while others can be implemented after the component installation phase. Author's take the help of UML diagrams as well as source code to show their work. Their argument is that the component should have customization properties in order to increase the reusability. In their proposed metrics author's used McCabe's Cyclomatic complexity and Alan's function points as the base to compute the complexity and reusability of a particular program or method.

Method:

Cho et al. categorised their quality estimation measures into three categories: Complexity, Customizability, and Reusability.

Complexity metrics: Author's proposed four classes of complexity metrics for components- plain, static, dynamic, and Cyclomatic.

Plain metrics- It is defined on the basis number of classes, abstract classes, interfaces, methods, complexities of individual classes, methods, corresponding weights, attributes and arguments.

In their work, author's identified two types of classes: internal and external classes. Internal classes are defined in the component whereas external classes are called from other components or libraries. Similarly there are two types of methods: internal and external methods. Internal methods are defined within the class whereas external methods are called from other classes. Weights are assigned to only internal classes and internal methods.

Static Complexity metrics- Static complexity is measured considering the internal structure of the component on the basis of associations among classes, as-

Component Static Complexity = Summation of (number of associations among classes * weight of corresponding association).

Five types of associations are identified and are assigned weight according to their precedence in order composition, generalization, aggregation, and dependency. These associations are computed two classes at a time.

Dynamic Complexity metrics- Dynamic complexity is measured by taking the number of messages passed between the classes into account, within the component, as-

Component Dynamic Complexity = Summation of (Number of messages * frequency of messages) + (summation of count of number of single parameter + Complexity of each message (summation of (number of complex parameters * weight of corresponding parameter)).

This metric is dynamic in nature since the number of parameters depends on the nature of execution.

Cyclomatic Complexity metric- It is defined with the help of source code developed. Author's used McCabe's Cyclomatic complexity to assess the complexity of each method existing in a class.

Customizability Metrics: Customizability is an attribute of a component that assures the level of reuse of that component. They identified three categories of customizable units in a method and arranged in their priority order as- attribute, behaviour, and workflow. Considering the level of complexity, author's assigned corresponding weights to the behaviour and workflow methods. To estimate the customization level, author's suggested a formula as-

Reusability metrics: Reusability metric is defined at two levels. First is at component level which assesses the reusability of a component in various applications and second is the reusability of components at the individual application level.

Key Findings:

- Defined metrics covers static as well as dynamic aspects of the component and the application, which are applicable to design and post implementation time of the development.
- As the value of plain complexity increases, the value of component Cyclomatic complexity increases. Dynamic complexity metrics exhibit more accurate results than static complexity metrics.
- Size, effort, cost and development time of component and component based applications can me measured early and easily in the development phase.

Metrics Used:

- McCabe's Cyclomatic complexity.
- Alan Albrecht's function point analysis.
- UML class diagrams, component diagrams, and deployment diagrams.
- Structure of the code

Factors affecting Interaction and Integration Complexity:

- Number of internal and external classes, internal and external methods, and In-out interfaces.
- Weights of internal classes, complex attributes, complex parameters, and methods.
- Number of associations and their weights.
- Number of messages, and their frequency.

Critique:

- Dynamic complexities are based on lines of code and function points. These metrics are have their own problems and are heavily criticized by practitioners.
- It is not clear that how the weights associated with different entities during complexity estimations will be computed or assigned.

F. Kenneth Morris's Complexity Metric

Kenneth Morris [14] proposed some object-oriented metrics to assess complexity and productivity metrics. Author's identified some complexity factors like Maintainability, Reusability, Extensibility, Testability, Comprehensibility, Reliability and Authorability, that they called "productivity impact variables". Morris proposed a complete set of nine eligible metrics for Methods, Class, Inheritance, Coupling and Cohesion.

G. Other Profound Complexity Metrics

Boehm [7] developed the 'object-point' metric through level of complexity of the amount of screenshots, reports and components. The level of complexities is categorized as simple, medium or difficult.

Chidamber and Kemerer's [15] proposed a metric suite for object-oriented software called as CK Metrics-suite. This metric suite is one of the most detailed and popular research works for object-oriented applications. Authors defined metric suite for complexity, coupling cohesion, depth of inheritance, and response set. These metric set are used to asses the complexity of an individual class as well as the complexity of the entire software system. In their metrics, Chidamber and Kemerer used Cyclomatic method for the complexity computation of individual classes.

Abreu and Rogerio Carapuca [16]-[18] proposed a metric set named 'Metrics for Object-Oriented Design'. In this metric suite, two fundamental properties of object-oriented programming are used, attributes and methods. Authors proposed metrics for the basic structural system of object-oriented idea as encapsulation, inheritance, polymorphism, and message passing. This suit consists of metrics for methods and attributes as assessment method for encapsulation.

Narasimhan et al. [19] suggested couple of metrics to assess the complexity of Component-Based Software. The packing density metric maps the count of integrated components, and the interaction density metric is used to analyse the interactions among components. They identified some constituents of the component in their work; these constituents include line of code, operations, classes, and modules. Authors also suggested a set of criticality criteria for component integration and interaction.

Vitharana et al. [20] developed a method for fabrication of components. Authors suggested some managerial factors like cost-efficiency; assembling easiness, customization, reusability, and maintainability. These are used to estimate technical metrics as coupling-cohesion, count, volume and complexity of components. They developed 'Business Strategy-based Component Design' model.

Rashmi Jain et al. [21] assess the association and mappings of cause-and-effect among the requirements of the system, structural design of the system and the complexity of the procedure of the systems integration. They argued the requirement of fast integration of components so that the complexity impact of integration on architectural design of components can be controlled. Authors identified 5 major factors to analyse the integration complexity of software system. Further these factors are divided into 18 sub-factors including commonality in hardware and software subsystems, percentage of familiar technology, physical modularity, level of reliability, interface openness, orthogonality, testability and so on.

Trevor Parsons et al. [22] proposed some specific dynamic methods for attaining and utilising interactions among the components in component-based development. They also proposed component-level interactions that achieve and record communications between components at runtime and at design time. For their work, authors used Java components.

Lalit and Rajinder [23] proposed a set of integration and interaction complexity metrics to analyse the complexity of Component-Based Software. They argue that complexity of interaction have two implicit features, first within the component, and second interaction from the other components. Their complexity metrics include percentage of component interactions, interaction percentage metrics for component integration, actual interactions, and total interactions performed, complete interactions in a Component-Based Software.

Some complexity assessment techniques for CBSE are on the basis of complexity properties including communication among components, pairing, structure, and interface [24]. The interaction and integration complexity measures available in the literature are explored considering the development paradigms like: Convention Software and Programs, Objet-Oriented Software, and Component-Based Software.

IV. PROPOSED INTERACTION COMPLEXITY ASSESSMENT TECHNIQUE

In this paper we propose a complexity computation method for component-based software based on inner and outer interactions. This technique is helpful to identify the number of actual interactions for those components whose source code may or may not be available.

A. Terminologies Used

First we define terminologies which are used to define the interaction metric.

Interaction-Graph: On the basis of communication among modules/components we draw an Interaction-Graph. We define an Interaction-Graph as a graph containing vertices and edges, where vertices represent modules/components and edges denote links among them, as shown in Fig. 1. Interaction-Graph depicts the flow and information among modules/components from source to destination.

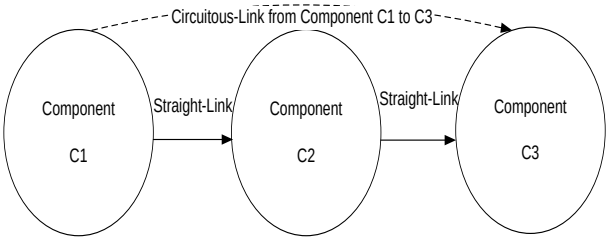

Fig. 1. Interaction-Graph for three components

Links: In this work we define two types of links for an Interaction-graph: Straight-links or Circuitous-links. These links decide the level or arity of interaction between two modules or components.

i. ***Straight-Links:*** Straight-links are the edges that connect two modules or components directly. Straight-link is shown in Fig. 1. From Component C1 to C2 there is a straight-link.

ii. ***Circuitous-Links:*** Circuitous-links are the edges that do not connect two modules or components directly. Circuitous-link includes two or more edges to connect two modules or components. Circuitous-links are shown in Fig. 1. From Component C1 to C3 there is a circuitous-link.

Interactions: This method defines two types of interactions:

i. ***Inner Interactions within a component (C_{in}):*** A component is made up of different constructs like simple statements, looping, branching and other similar constructs. Inner interaction defines the number of interactions made by the inner constructs of the component. Inner interactions are intra-component interactions. These constructs may be straight or circuitous linked, as shown in Fig.

2.

ii. ***Outer Interactions among Components (C_{out}):*** Defines the number of interactions shared by the two components. Outer interactions are inter-component interactions, as shown in Fig. 3.

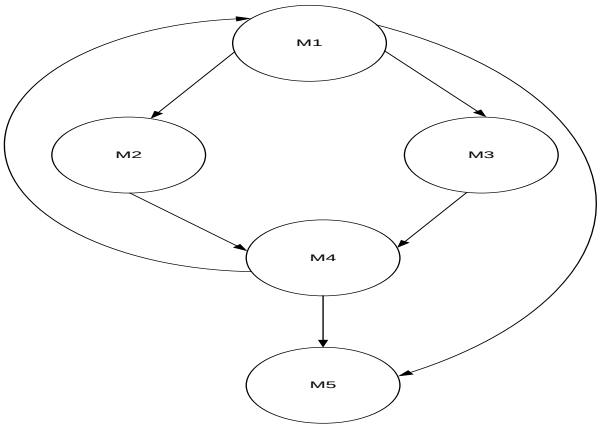

Fig. 2. Interaction-Graph for inner interactions of a component

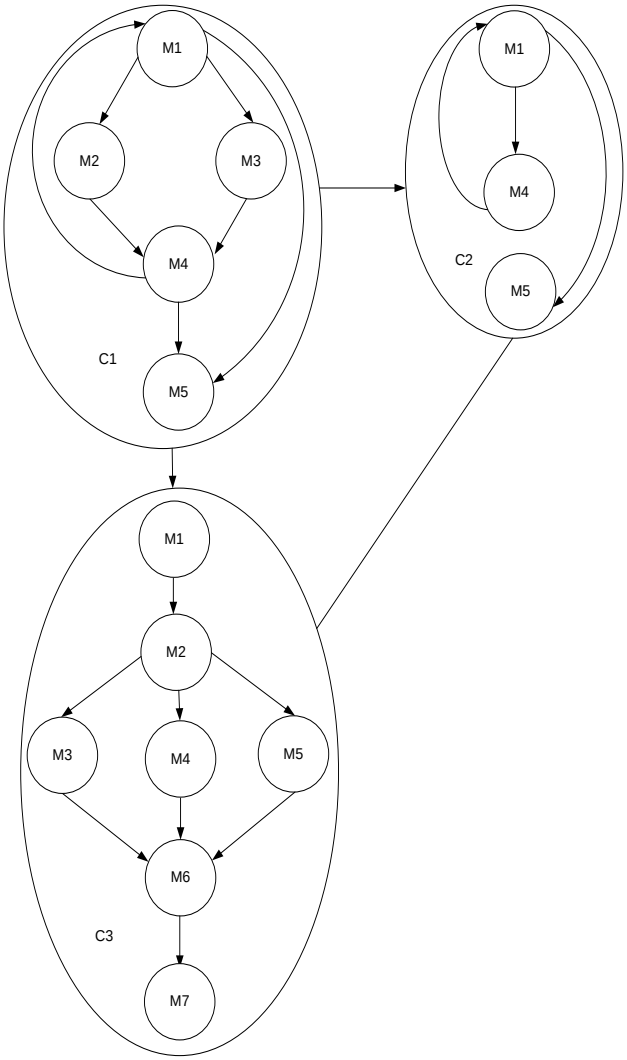

Fig. 3. Interaction-Graph of Example Case Study exploring Outer Interactions of Components

Straight-Circuitous-Link Matrix:

Straight-circuitous-link matrix is a row column matrix containing total number of rows and columns as the number of components in the interaction-graph, as shown in Table 1. If there is a straight-link between two components then we put '1' in the matrix. If the link is circuitous then the total number of edges between these module/components will be placed. These values are shown in Table 1.

i. ***Straight-Link value among modules/components:*** There is a straight-link between C1 and C2, therefore its corresponding value in the matrix is '1'.

ii. ***Circuitous-Link value among modules/components:*** Component C1 and C3 are linked circuitously. There is a straight-link between component C1 and C2, so its value is '1'. Similarly components C2 and C3 are straight linked; therefore its corresponding value is also '1'. Hence C1 to C3, there is a circuitous-link; its value is 1+1=2.

Table 1 defines the Straight-circuitous-Links for the individual component shown in Fig. 2. M1, M2, M3, M4, and M5 represent modules and constructs of the component.

Table 1. Straight-Circuitous-Link Matrix for Component shown in Fig. 2

Modules	M 1	M 2	M 3	M 4	M 5	Total interactions in which a module is involved
M1	1	1	1	2	3	8
M2	0	1	0	1	1	3
M3	0	0	1	1	2	4
M4	1	0	0	1	1	3
M5	0	0	0	0	1	1
						19

Table 2 shows the Straight-Circuitous-Link matrix for example case study defined in Fig. 3. C1, C2, and C3 represent components involved in the case study. Each component consists of internal constructs and has inner-interactions.

Table 2. Straight-Circuitous-Link Matrix for Case Study shown in Fig. 3

Components/ Modules	C1	C2	C3	Total interactions in which a component is involved
C1	1	1	1	3
C2	0	1	1	2
C3	0	0	1	1
				6

B. Calculation of Interactions

In this section we propose some computation metrics to assess the interactions among modules/components.

Actual Interactions of Component (CTotal):

An actual interaction of component is defined as the number of inner interactions made by a particular component. We compute actual interaction as defined in Equation (1),

Actual Interactions of Component =

$$\sum_{i=1}^{n} C_{in} + C_{out} \quad (1)$$

Where 'm' represents the number of modules in the component, and C_{in} represents the total number of inner interactions of the component.

From Table 1 we calculate the number of Inner-interactions of Component = 17

Actual Interactions of Component-based software (CBSTotal):

Actual Interactions of Component-based software is defined as the number of inner-interactions made by the particular component and the total number of outer-interactions made by all the components in the CBS application [25]. We assess actual interactions of CBS application as defined in Equation (2),

Actual Interactions of CBS =

$$\sum_{i=1}^{n} C_{in} + C_{out} \quad (2)$$

Where, CBS_{Total} defines the actual interactions of component-based software, 'i' represents the number of components in the CBS application, C_{in} and C_{out} defines total inner and outer interactions of individual components respectively.

Average Number of Interactions of Components in Component-based software (CBSAvg):

Average number of interactions of made by components in component-based software is defined as the ratio of Actual Interactions of CBS and the total number of components involved in the CBS application. We compute average interactions of components in CBS as defined in Equation (3),

$$\text{Average Interactions of CBS} = \frac{\sum_{i=1}^{n} C_{in} + C_{out}}{\text{Total number of Components in CBS}} \quad (3)$$

Where, CBS_{Avg} is the average interactions of component-based software, 'i' represents the number of components in the CBS application, C_{in} and C_{out} defines total inner and outer interactions of individual components respectively.

C. Calculation of Interactions

To illustrate our proposed metrics we define an exemplar case study containing three components, C1, C2, and C3. Each component have inner interactions and outer interactions containing straight-links as well as circuitous-links. Table 3 describes the Straight-Circuitous-Link Matrix for Component C1 of case study defined in Fig. 3.

Table 3. Straight-Circuitous-Link Matrix for Component C1

Modules	M 1	M 2	M 3	M 4	M 5	Total interactions in which a module is involved
M1	1	1	1	2	3	8
M2	0	1	0	1	2	4
M3	0	0	1	1	2	4
M4	1	0	0	1	1	3
M5	0	0	0	0	1	1
						20

Table 4 describes the Straight-Circuitous-Link Matrix for Component C2 of case study defined in Fig. 3.

Table 4. Straight-Circuitous-Link Matrix for Component C2

Modules	M 1	M 2	M 3	Total interactions in which a module is involved
M1	1	1	1	3
M2	1	1	1	3
M3	0	0	1	1
				7

Table 5 describes the Straight-Circuitous-Link Matrix for Component C1 of case study defined in Fig. 3.

Actual interactions of Components:

From Table 3, 4, and 5 we compute the Inner-interactions of each component of cases study defined in Fig. 3.

Inner-interactions of Component C1 = 20

Inner-interactions of Component C2= 7

Inner-interactions of Component C3= 39

Actual interactions of Component-Based Software:

Therefore actual interactions made by the CBS application defined in Fig. 3 is,

$\sum_{i=1}^{n} C_{in} + C_{out}$ = 20 + 7 + 39 + 6 = 72

Average number of interactions of components in Component-Based Software:

Average Interactions of CBS = $\frac{\sum_{i=1}^{n} C_{in} + C_{out}}{\text{Total number of Components in CBS}}$

= 72/3

= 2

Table 5. Straight-Circuitous-Link Matrix for Component C3

	M 1	M 2	M 3	M 4	M 5	M 6	M 7	Total interactions in which a module is involved
M 1	1	1	2	2	2	3	4	15

M 2	0	1	1	1	1	2	3	9
M 3	0	0	1	0	0	1	2	4
M 4	0	0	0	1	0	1	2	4
M 5	0	0	0	0	1	1	2	4
M 6	0	0	0	0	0	1	1	2
M 7	0	0	0	0	0	0	1	1
								39

V. CONCLUSION

Methods and metrics proposed so far in the literature are defined on the basis of interactions among instructions, operations, procedures, and functions of individual and standalone programs and codes. These metrics are appropriate for small-sized codes. Some measures are also defined for object-oriented software, but for CBSE applications these methods are not inadequate. In the CBSE, components have connections and communications with each other to exchange services and functionalities. Interaction edges are used to denote the connections among components. In this work we define some simple metrics to assess the interaction of component-based software. Metrics defined in this work consider the individual interactions of components as well as inter-component interactions. These metrics are helpful to explore the non-functional attributes of components and component-based software.

REFERENCES

1. B. W. Boehm, M. Pendo, A. Pyster, E. D. Stuckle, and R. D. William, "An Environment for Improving Software Productivity," *IEEE Computer*. 1984.
2. Pressman Roger, *Software Engineering A practitioners Approach*. 6th ed, TMH International edition, 2005.
3. Wake, and S. Henry, "A Model Based on Software Quality Factors which Predict Maintainability", in *Proceedings of the Conference on Software Maintenance*. 1988, pp. 382-387.
4. R. Basili, and D. H. Hutchens, "An Empirical Study of a Syntactic Complexity Family," *IEEE Trans. on Software Engineering*. vol. 9, no. 6, 1983, pp. 664-672.
5. M. H. Halstead, *Elements of Software Science*. New York, Elsevier North Holland, 1977.
6. Alan Albrecht, and J. E. Gaffney, "Software Function Source Line of code and Development Effort Prediction: A Software Science Validation," *IEEE Trans. on Software Engineering, SE-9.* 1983, pp. 639-648.
7. B. Boehm, "Anchoring the Software Process," *IEEE Software.* vol. 13, no. 4, 1996, pp. 73-82.
8. M. M. Lehmam, and L. A. Belady, "Program Evolution - Processes of Software Change," 1985.
9. Usha Kumari, and S. Bhasin, "A composite complexity measure for component-based systems," *ACM SIGSOFT Software Engineering Notes*. vol. 36, no. 6, 2011.
10. T. McCabe, "A complexity measure," *IEEE Trans. on Software Engineering.* vol. 2, no. 8, 1976, pp. 308–320.
11. S. Henry, and D. Kafura, "Software Structure Metrics Based on Information Flow," *IEEE Trans. on Software Engineering*. vol. 7, 1981, pp. 510-518.
12. http://en.wikipedia.org/wiki/complexity.
13. E. S. Cho, M. S. Kim, and S. D. Kim, "Component Metrics to Measure Component Quality, in *Proceedings of the Eighth Asia-Pacific on Software Engineering Conference (APSEC '01).* IEEE Computer Society, Washington, DC, USA, 2001, pp.419-426.
14. K. Morris, *Metrics for Object Oriented Software Development.* Masters thesis, M.I.T., Sloan school of management, Cambridge, MA, (1989).
15. S. Chidamber, and C. Kemerer, "A Metrics Suite for Object -Oriented Design," *IEEE Trans. on Software Engineering*. vol. 20, no. 6, 1994, pp. 476-493.
16. F. B. Abreu, and Rogerio Carapuca, "Object-Oriented Software Engineering: Measuring and Controlling the Development Process," in *Proceedings of the 4th International Conference on Software Quality*. McLean, VA, USA, 1994. pp. 3-5.
17. F. B. Abreu, "Design Metrics for Object-Oriented Software System," in *Proceedings. Workshop on Quantitative Methods ECOOP*. 1995, pp. l-30.
18. F. B. Abreu, and W. Melo, "Evaluating the Impact of Object- Oriented Design on Software Quality," in *Proceedings of the 3rd International Software Metrics Symposium.* Berlin, Germany, 1996.
19. V. L. Narasimhan, and B. Hendradjaya, "Theoretical Considerations for Software Component Metrics," *Trans. on Engineering, Computing and Technology*, vol. 10, 2005, pp. 169-174.
20. Padmal Vitharana, Hemant Jain, and Fatemeh Mariam, "Strategy-Based Design of Reusable Business Components," *IEEE Trans. on Systems, Man, and Cybernetics—PART C: Applications and Reviews*. vol. 34, no. 4, 2004.
21. Rashmi Jain, Anithashree Chandrasekaran, George Elias, and Robert Cloutier. "Exploring the Impact of Systems Architecture and Systems Requirements on Systems Integration Complexity," *IEEE Systems Journal*, vol. 2, no. 2, 2008.
22. Trevor Parsons, Adrian Mos, Mircea Trofin, Thomas Gschwind, and John Murphy, "Extracting Interactions in Component-Based Systems," *IEEE Trans. on Software Engineering*, vol. 34, no. 6, 2008.
23. Latika Kharb, and Rajender Singh, "Complexity Metrics for Component-Oriented Software Systems," *ACM SIGSOFT Software Engineering Notes*, vol. 33, no. 2, 2008.
24. Umesh Tiwari and Santosh Kumar, "Cyclomatic complexity for component based software," A*CM SIGSOFT Software Engineering Notes.* vol. 39, pp. 1-6, 2014.
25. Umesh Tiwari and Santosh Kumar "In-Out Interaction Complexity Metrics for Component-Based Software," *ACM SIGSOFT Software Engineering Notes.* vol. 39, 2014.

AUTHORS PROFILE

Dr. Umesh Kumar Tiwari is working as an Associate Professor in Department of Computer Science and Engineering in Graphic Era Deemed to be University, Dehradun. He had received his Ph.D. in 2016. He has more than 12 years of experience in teaching/research of UG and PG level degree courses as a Lecturer/Assistant Professor/ Associate Professor in various academic/research organizations. He is supervisor of 02 PhD and 5 M. tech students who are working on specific domains of software engineering and network security. His research is on multidisciplinary topics and he has published 17 journal papers in reputable international and national journals, and 12 conference papers in reputed international and national conferences. His research interests are Wireless Communication Networks, Network Security, and Software Engineering topics with improved modeling, interaction-integration complexities, testing and reliability models.

Dr. Santosh Kumar had received his Ph.D. from IIT Roorkee (India) in 2012, M. Tech. (CSE) from Aligarh Muslim University, Aligarh (India)in 2007 and B.E. (IT) from C.C.S. University, Meerut (India) in 2003. He has more than 13 years of experience in teaching/research of UG (B. Tech.) and PG (M.Tech.) level courses as a Lecturer/Assistant Professor/ Associate Professor in various academic /research organizations. He has supervised 01 Ph.D. Thesis, 20 M.Tech. Thesis, 18 B.Tech projects and presently mentoring 06 Ph.D students, 03 M.Tech students and 04 B.Tech. students. He has also completed a consultancy project titled "MANET Architecture Design for Tactical Radios" of DRDO, Dehradun in between 2009-2011.He is an active reviewer board member in various national/International Journals and Conferences. He has memberships of ACM (Senior Member), IEEE, IAENG, ACEEE, ISOC (USA) and contributed more than 46 research papers in National and International Journals/conferences in the field of Wireless Communication Networks, Mobile Computing and Grid Computing and software Engineering. Currently holding position of Associate professor in the Graphic Era Deemed to be University, Dehradun (India). His research interest includes Wireless Networks, MANET, WSN, IoT, and Software Engineering.