Indian Journal of Science and Technology, *Vol 11(32), DOI: 10.17485/ijst/2018/v11i32/123093, August 2018*
ISSN (Print) : 0974-6846
ISSN (Online) : 0974-5645

Complexity Metrics for Component-based Software Systems: Developer Perspective

Sellami Ali[1] , Majdi Abdellatief[1,2*], Mohamed Ahmed Elfaki[1] and Abubaker Wahaballa[2,3]

[1]College of Computing and Information Technology, Shagra University, Riyadh, Kingdom of Saudi Arabia; sellami2003@hotmail.com, malfaki@su.edu.sa
[2]MTC College, Sudan Technological University, Sudan, 2081 Khartoum; khwaja@su.edu.sa, khwaja24@yahoo.com
[3]Arab East Colleges, Computer Science and Technology, Riyadh, Kingdom of Saudi Arabia; wahaballah@edu.sa

Abstract

Background: A Component-Based Development (CBD) is an integration centric system focusing on assembling individual components in order to build a software system. Most of the existing CBD metrics rely on parameters that are too difficult to measure in practice due to the component's internal elements may not be visible to developers or testers. **Objective:** We proposed two suite of metrics to measure the structural complexity and interaction complexity of Component-Based Software System (CBSS) from perspective of component developer. **Methods:** Based on the analysis of the component specification, the elements of interface which includes properties, methods and events are measured. The proposed metrics quality is evaluated from a mathematical perspective using BMB properties. **Finding:** The theoretical evaluation results indicated that the proposed metrics are valid internal measures. The proposed metrics are useful in understanding and identifying the areas in the design where improvements are likely to have a high attention. Thus, the proposed metrics appear promising as a means of capturing the quality of the CBSS design in question. **Application/Improvements:** It has been widely reported that lower complexity is believed to provide advantages such as lower maintenance time, easier to test, highly reusable and easier to understand.

Keywords: Component, Complexity, Metrics, CBSS

1. Introduction

Using reusable software components, developers can "drag and drop" components onto a system instead of writing code. In actual practice of the CBD, we do not need to write even a single line of code. Instead, we use the techniques of visual programming such as clicking, dragging and dropping of components. However, nowadays, the current practice of CBD is a mixture of the two developments styles: visual programming which allows developing a specific type's component, while conventional programming allows us to specify the behavior of business process or component.

Considering above introduction, many people think that CBSS construction can be designed like building by putting pieces together, but this is hardly convincing. The truth is that a CBSS is much more complex a system consisting of many complex processes. In fact, CBSS is developed by many people and different programming languages leading to structure, interaction, and complexity issues. A few researches on CBSS structural complexity have been carried out in recent years, and contributions to the area are reported in[1–6]. A Study by Narasimhan[7], reported that some of these metrics rely on parameters that are too difficult to measure in practice or could never be measured. This is due to the component's internal structure may not be visible to developers or testers. Another limitation of these metrics is that they all only consider one individual factor as a means to measure component complexity. The aim of this paper is to propose a new metric to measure CBSS structural and interaction attributes. The proposed metric is used at the design phase of CBSS development process.

*Author for correspondence

2. Software Component Background

Several definitions of a component are given in[8–11], each of which stated different characteristics of software components. However, the whole software engineering community agrees that, a component is a provider of service, and it is a composed of two parts: an interface (declaration part) and a body (implementation part). The interface contains the resources and elements (e.g. properties, methods and events) that make the component visible for CBSS developers and tester. The body contains the implementation detail that is not to be visible to CBSS developers. Figure 1, provides a simple model for component, taking into account the overall interface elements such as Properties, Methods and Events (PME). It shows that a component has a required interface and a provided interface. Those interfaces have properties to control the visible part of component. A summing that method can provide or require functions, and events can send and receive a notification.

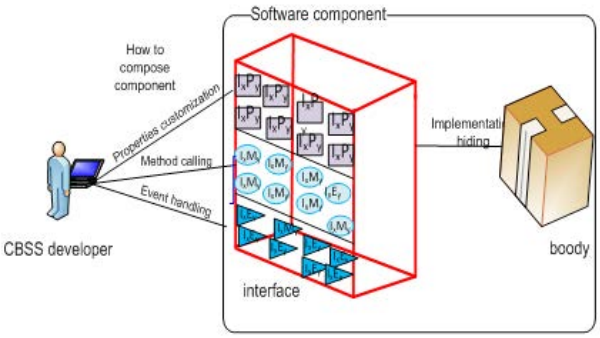

Figure 1. A simple model of component.

3. CBSS Structure

Generally, structural complexity of traditional software development is defined as the difficulty to analyze source code, revise and maintain its modules. In object-oriented development, it is defined as the difficulty of analyzing both source code and coupling between classes and objects. Whereas, in CBSS, due to its black box nature, structural complexity can be defined as the difficulty to customize, substitute, add or remove components at design time or run-time. Customization can be performed through the analysis of the structure of interface which includes properties, methods and events. Addition and removal can be performed through the analysis of interaction complexity.

In the high-level design, a CBSS seems as a collection of components, properties, methods and events. Usually, an application is constructed by customizing properties, handling event and adapting method calling using the properties window. In other words, at high-level design time, only the component and interface elements (PME) are visible, but nothing is known about the inner workings of the body of components (implementation part) in order to make use of it. Therefore, the structure of CBSS can be identified by the set of components and the set of connections between PME of source component and PME of target components. Intuitively, different types of structures of the same system will certainly lead to different values of structural complexity.

With respect to the paper aims, we reason about interaction and the structure of interface in isolation as mainly contributors to the complexity of the CBSS. This view is consistent with the view of Salman[5]. Thus, the solutions to this design complexity center on adapting the connections between components and analyzing interface structure carefully. At this point, for better understanding of interface specification with respect to the goal of this paper, all its constituents are investigated in details in the following sub-section.

3.1 Interface

An interface is a framework defining a set of services that a component must implement, and that supports communication between components. For example, Java Bean and .NET interfaces are seen as an abstract classes with public abstract list of Properties, Methods, and Events (PME) that can be implemented somewhere (component body).

Example Interface definition in C# language
Public interface ISampleInterface
{
int Add (object value); // method declaration consists of return's type, name and parameters.

Public **string** file name {get; set ;} // property declaration consists of accessibility type, property type and property name.

Public event EventHandler Click; // an event declaration includes accessibility type, event keyword, event type and event name}

3.2 Propoery

From a developer viewpoint, a property looks like a variable, serves as an object-oriented attribute, and acts like a method[12]. CBSS developer may change the property value through visual interface or using code, to customize and configure component at run time[13–16]. For example, think about how to control television, digital and remote-control machines. They share properties such as: an on/off property, volume property, channel property and a color property. Thus, how properties are implemented considerably affects the complexity of properties. The number of properties involved is an indication of how much time and effort is required to integrate and maintain a CBSS.

3.3 Method

A method is a name given to function in interface, whereas a property is a name given to a variable in interface. The method represents the service or functionality that the component provides. It is defined in terms of parameters types constituting the signature of the method. The method realizes the dynamic behavior of a system in the form of explicit operation invocation or message passing which usually is called proactive control.

3.4 Event

An event is a message sent from one component to another component, if specific action happened such as birthday or button click. It is another form of control used to realize the dynamic behavior of the system[12]. Thus, components use events to interact with other components in a system. Events are similar to methods but semantically they stand for call initiated by external causes in contrast to calls made under program control. Other than this, methods and events are similar in the way they are represented[17].

3.5 CBSS INTERNATION

The interaction of CBSS is specified by connecting components with each other using properties, methods and events which are exposed through component interfaces. As long as component-software interaction occurs on the connections specified between components, those relationships will be called interactions and will be used to define metrics that capture the complexity of CBSS. We define an interaction as "*an action between two or more software elements*"[18]. Taking into consideration the overall interface characterization of a component, we identify the following types of interactions that mainly contribute towards the interaction complexity of CBSS:

3.6 Event to Property Interation

In .NET and Java-Bean, using the properties window, CBSS developers can link an event from the source component to manipulate the properties of the target components as they see fit. There are three main operations that can be associated with an event-property interaction: getting the value of property, setting a new value and editing the property. For example, you would need to reset the property value to an initial value. To do this, you will click the reset button which will call an event method. The important thing in these operations is that the signature of the source event must match the signature of the target property.

3.7 Event to Methods Interaction

A component can offer services (or function) to the other component. It can also require service from the other component. A method interaction is specified by coupling an event of source component with a method of the target component. The basic operation of event-method connection is the explicit operation invocation or message passing.

3.8 Event to Event Interaction

An event interaction is specified by coupling an event of the source component to an event of the target component. An event can send notification to only one component or to multiple components at a time. The main operation supported by an event of the source component is the subscription and un-subscription of an event notification.

4. Metrics Definitions

4.1 Structural Complexity Measurements

The structure of a component is defined as the set of properties it has, the set of methods (or services / functions) it defines, and the set of events it creates.

Total Number of Elements (TNE) is the total count of properties, methods and events declared in a component.

$$TNE = \sum (Pi + Mi + Ei) \quad (1)$$

where:

Pi is a count of the number of properties
Mi is a count of the number of methods
Ei is a count of the number of events.

The hypothesis is that the larger the number of properties, methods and events in a components, the more likely to contain lower level functionality, and the more difficult to understand, which in turn implies the more effort to integrate, test, and maintain[9,19]. Developers should minimize the structure of the CBSS by eliminating unnecessary elements; make it easy to understand, through customizing functionalities, removing redundant properties and events from a component.

From the above metric we derive other three metrics as follows:

Property Density (PD): This metric is the ratio of the number of properties to the total number of elements in a component:

$$\mathbf{PD} = \frac{NP}{TNE} \quad (2)$$

Method Density (MD): This metric is the ratio of the number of methods to the number of elements in a component:

$$\mathbf{MD} = \frac{NM}{TNE} \quad (3)$$

Event Density(ED): This metric is the ratio of the number of events to the total number of elements in a component:

$$\mathbf{ED} = \frac{NE}{TNE} \quad (4)$$

At this point, we can ask what the best density of PME for the component is? Is there a best density? We believe the answer is yes, although there is no such thing as a perfect reference. This is because it's varies based on the domain and the component developer skills. But, we must know what is too high density or too low density based on statistical information or logical arguments. For example, fasion[9] claimed that "*a good heuristic might be to shoot for components with the following number of elements: 10 properties, 10 methods and 10 events*". Therefore, the metric value can be useful semantically.

4.2 Interaction Complexity Measurements

The interaction of CBSS is designated by connecting components with each other using properties, methods and events. The property, method and events are defined in terms of parameters types constituting their "signature". A component can be connected with target components directly if and only if its provided interface is exactly matched with the required interface of a target component[20].

Based on the above interpretation, we define metrics in terms of signature similarity. Two interfaces are similar if the union of the sets of PME signatures used is substantially matched.

Matched Property Count (MPC): This metric counts the number of properties in the target component whose signature matches the types of properties in the source components.

$$MPC = IC_{1P} \bigcap IC_{2P} \quad (5)$$

Matched Methods Count (MMC): This metric counts the number of methods in the target component whose signature matches the types of methods in source components.

$$MMC = IC_{1M} \bigcap IC_{2M} \quad (6)$$

Matched Events Count (MEC): This metric counts the number of events in the target component whose signature matches the types of events in the source components.

$$MEC = IC_{1E} \bigcap IC_{2E} \quad (7)$$

The underlying theory of these metrics is that a component with the greater number of MPC, MMC and MEC indicates a higher optionality of interaction paths, and the communication can be distributed and controlled as we need throughout the application which in turn proves the easier to integrate and reuse. In contrast, a component with a lower number of MPC, MMC and MEC could result in the difficulty of integration and maintenance process, because it will be necessary to create an adapting shell around the component[20,21].

From metrics above we derived other three metrics as follows:

Matched Property Ratio (MPR): This metric is the ratio of MPC and total number of properties in application

$$MPR = \frac{MPC}{NP} \quad (8)$$

Matched Methods Ratio (MMR): This metric is the ratio of MPC and total number of methods in application

$$MMR = \frac{MMC}{NM} \quad (9)$$

Matched Events Ratio (MER): This is a derivative metric, which is the ratio of MEC and total number of methods in application

$$MER = \frac{MEC}{NE} \quad (10)$$

5. Theoretical Validation of Proposed Metrics

The main goal of this section is to evaluate the proposed metrics quality from mathematical perspective. Several authors have introduced desirable properties for software measurements which used to validate existing or newly proposed software measurmes[22–26]. As has been noted the properties given by Brain, Morasca and Basili (BMB)[22] are reported to be more practical and more popular than others in term of Google Scholar citation. The reader should refer to the description of properties of BMBs to get a better understanding of the theoretical validation of proposed metric system. The two set of metrics we proposed are indeed complexity metrics. Therefore, we need to validate them against complexity properties which are Non-negativity, Null value, Monotonicity, Symmetry and Disjoint module additivity.

5.1 Structural Complexity Metrics Evaluation and Discussion

Non-negativity:

1-The TNE metric is obtained by counting the total count of properties, methods and events declared in a component, so it can be zero (null) or positive, but cannot be negative.

2- The PD, MD and ED metrics is a ratio of non-negative numbers (i.e. NP, NM, NE and TNE), so it cannot be negative in value

Null value:

1-We expect the TNE to be null when a component interface does not contain any property, method or event

2- We expect the PD, MD and ED values to be null, when NP, NM and NE values are null, respectively.

Monotonicity:

1- The TNE of a component is a sum of the TNE of individual interfaces that may make it up. Consequently, the sum of the TNE of any two of its interfaces cannot be more than the TNE of components.

2- PD, MD and ED of component is a sum of PD, MD and ED of its individual interfaces respectively. Consequently, the sum of the PD, MD and ED of any two of its interfaces cannot be more than the component's PD, MD and ED respectively.

Symmetry:

1- In order to integrate two interfaces together we have to link each property, methods and events of the first interface and the other the interface. Consequently, this property holds.

2- PD, MD and ED are metrics used as relationship indicator between interfaces. Thus, this property holds.

Disjoint module additivity:

1- Given two disjoint interfaces (or non-connected interfaces) the TNE obtained by integrating two the interfaces will be $TNE_{(1)} + TNE_{(2),}$ thus this property holds.

2- Given that a component 'C' is formed by two single and non- connected components 'A' component 'B', the following statement hold:

PD (C) = PD (A) + PD (B)
MD (C) = MD (A) + MD (B)
ED (C) = ED (A) + ED (B)

5.2 Integration Complexity Metrics Evaluation and Discussion

Non-negativity:

1- The MPC metric is obtained by counting matched signature of properties in the source components and target component, so it can be zero (null) or positive, but cannot be negative. In the same way MMC and MEC can be zero (null) or positive, but cannot be negative.

2- The MPR, MMR and MER metrics is a ratio of non-negative numbers (i.e. MPC, MMC, and MEC), so it cannot be negative in value

Null value:

1-We expect the MPC, MMC and MEC to be null when a component interface does not contain any matched signature of property, method or event respectively.

2- We expect the MPR, MMR and MER values to be null, when MPC, MMC, and MEC values are null, respectively

Monotonicity:

1- The MPC of a component is a sum of the MPC of individual interfaces that may make it up. Consequently, the sum of the MPC of any two of its interfaces cannot be more than the MPC of the components. In the same way, this property holds for MMC and MEC.

2- MPR, MMR and MER of component is a sum of MPR, MMR and MER of its individual interfaces respectively. Consequently, the sum of the MPR, MMR and MER of any two of its interfaces cannot be more than the component's MPR, MMR and MER respectively.

Symmetry:

1- In order to integrate two interfaces together we have to count the matched signature of properties, methods and events between interfaces. Consequently, this property holds.

2- MPR, MMR and MER are metrics used as relationship indicator between interfaces. Thus, this property holds.

Disjoint module additivity:

1- Given two disjoint interfaces (or non-connected interfaces) the MPC obtained by integrating two interfaces will be $MPC_{(1)} + MPC_{(2)}$, thus this property hold.

2- Given that a component 'C' is formed by two single and non- connected components 'A' component 'B', the following statement hold:

MPR (C) = MPR (A) + MPR (B)
MMR(C) = MMR (A) + MMR (B)
MER (C) = MER (A) + MER (B)

6. Conclusion

The motivation of this paper is that the measurement based on the interface specification is suitable to characterize and evaluate the complexity of CBSSs. This is because, the internal of components are hidden and are unreachable except via abstract interfaces. This imposes difficulties on sufficient evolution of an integrated CBSS. In this paper, we proposed a suite of metrics to measure the structure and interaction of CBSS. We proposed two suite of metrics to measure the structural complexity and interaction complexity of Component-Based Software System CBSS. Results confirmed that the new metrics are theoretically valid and structurally sound since they satisfied the properties in their respective categories. Applying these measurements to the CBSS design can identify the complexity of the design, and give unbiased evaluation of the components. The proposed metrics can be used to identify complex components and/or critical components. Complex and/or critical components design would potentially take longer time to develop and substantial testing effort than a simple one. Therefore, developers, testers, with better experience should be assigned to integrate and test critical components.

7. References

1. Kumar P, Tomar P. Design of dynamic metrics to measure component based software. Computing, Communication and Automation (ICCCA), International Conference; 2017. p. 753–7.
2. Kharb L, Singh R. Complexity metrics for component-oriented software systems. SIGSOFT Software Engineer Notes. 2008; 33(2):1–3. https://doi.org/10.1145/1350802.1350811 https://doi.org/10.1145/1350802.1350810
3. Mahmood S, Lai R. A complexity measure for UML component-based system specification. Software Practice and Experience. 2008; 38(2):117–34. https://doi.org/10.1002/spe.769
4. Narasimhan L, Hendradjaya B. Some theoretical considerations for a suite of metrics for the integration of software components. Information Sciences. 2007; 177(3):844–64. https://doi.org/10.1016/j.ins.2006.07.010
5. Salman N. Complexity metrics AS predictors of maintainability and integrability of software components. Journal of Arts and Sciences; 2006. p. 39–50.
6. Brosig F, Meier P, Becker S, Koziolek A, Koziolek H, Kounev S. Quantitative evaluation of model-driven performance analysis and simulation of component-based architectures. IEEE Transport Software Engineering. 2015; 41(2):157–75. https://doi.org/10.1109/TSE.2014.2362755
7. Narasimhan VL, Parthasarathy PT, Das M. Evaluation of a suite of metrics for Component Based Software Engineering (CBSE). Issues in Informing Science and Information Technology. 2009; 6:731–40. https://doi.org/10.28945/1093
8. Crnkovic I, Larsson M. Building reliable component-based software systems. London Artech House. Estublier J, Favre JM. Component model and technology. Building reliable

component-based software system (). London: Artech House; 2002. PMid:12476477
9. Faison T. Component-based development with visual C#. New Yourk. Hungry Minds; 2002. PMid:12526889
10. Gill NS, Grover PS. Component-based measurement. Few useful guidelines. SIGSOFT Software Engineering Notes. 2003; 28(6):4–4. https://doi.org/10.1145/882240.882255 https://doi.org/10.1145/966221.966237
11. Szyperski C. Component software: Beyond object oriented programming (Second Edition ed.). New York. Addison Wesley; 2002.
12. Sharp J. Microsoft visual C#. Step by step Microsoft Press; 2008.
13. Gill NS, Grover PS. Few important considerations for deriving interface complexity metric for component-based systems. SIGSOFT Software Engineering Notes. 2004; 29(2):4–4. https://doi.org/10.1145/979743.979758
14. Han J. A comprehensive interface definition framework for software components. Proceedings of the Fifth Asia Pacific Software Engineering Conference; 1998. p. 110. https://doi.org/10.1109/APSEC.1998.733601
15. Sharma A, Kumar R, Grover PS. Empirical evaluation and validation of interface complexity metrics for software components. International Journal of Software Engineering and Knowledge Engineering. 2008; 18(7):919–31. https://doi.org/10.1142/S0218194008003957
16. Washizaki H, Yamamoto H, Fukazawa Y. A metrics suite for measuring reusability of software components. Proceedings of the 9th International Symposium on Software Metrics; 2003. p. 1– 211. https://doi.org/10.1109/METRIC.2003.1232469
17. Cesare SD, Lycett M, Macredie RD. Development of component-based information system. New Delhi. Prentice Hall of India; 2006. p. 1–239.
18. Heineman G, Councill W. Component definition. In B. Councill, G. Heineman (Eds.). Component-based software engineering Putting pieces together () Addison Wesley; 2001.
19. Abdellatief M, Sultan ABM, Ghani AAA, Jabar MA. A mapping study to investigate component-based software system metrics, Journal of System Software. 2013; 86(3):587–603. https://doi.org/10.1016/j.jss.2012.10.001
20. Abdellatief M, Sultan ABM, Ghani AAA, Jabar MA. Component-based software system dependency metrics based on component information flow measurements in The Sixth International Conference on Software Engineering Advances ICSEA2011 Barcelona, Spain; 2011. p. 76–83.
21. Abdellatief M, Sultan ABM, Ghani AA, Jabar MA. Multidimentional size measure for design of component-based software system, IET Software. 2012; 6(4):350–7. https://doi.org/10.1049/iet-sen.2011.0122
22. Briand LC, Morasca S, Basili VR. Property-based software engineering measurement. IEEE Transaction Software Engineering. 1996; 22(1):68–86. https://doi.org/10.1109/32.481535
23. Weyuker E.J. Evaluating software complexity measures. IEEE Transaction Software Engineering. 1988; 14(9):1357–65. https://doi.org/10.1109/32.6178
24. Zuse H. Software complexity measures and methods. Walter deGruyter; 1991.
25. Kitchenham B, Pfleeger SL, Fenton N. Towards a framework for software measurement validation. IEEE Transaction Software Engineering. 1995; 21(12):929–44. https://doi.org/10.1109/32.489070
26. Tian J, Zelkowitz MV. A formal program complexity model and its application. Journal of System and Software. 1992; 17(3):253–66. https://doi.org/10.1016/0164-1212(92)90114-Y