A Theoretical Validation of Component Point

Thareendhra Wijayasiriwardhane[1], Richard Lai[2*]

[1] Faculty of Science, University of Kelaniya, Sri Lanka.
[2] Department of Computer Science and Information Technology, La Trobe University, Australia.

* Corresponding author. Email: r.lai@latrobe.edu.au
Manuscript submitted Number 24, 2018; accepted January 12, 2019.
doi: 10.17706/jsw.14.1.1-12

Abstract: The system-level size measures are important in software project management as tasks such as planning and estimating the cost and schedule of software development can be performed more accurately when a size estimate of the entire system is available. However, due to the black-box nature of software components, traditional software measures are not adequate for Component-Based Systems (CBS). We have developed a Function Point (FP) like measure, named Component Point (CP), for measuring the system-level size of a CBS specified in Unified Modeling Language. In this paper, we present a theoretical validation of the CP measure using mathematics and show that not only the CP measure holds all the mathematical conditions necessary for a size measure, but it can also be used in Component-Based Software Development (CBSD) in a similar way that FP and its extensions are used in other software development paradigms.

Key words: Component-based systems, component point, software sizing, mathematical validation.

1. Introduction

Developing large and complex software systems remains a costly and a somewhat unpredictable business, even today. It is reported that 52% of software projects failed to deliver on schedule, within budget and thereby with the required functionality, while 19% of projects were totally abandoned [1]. This clearly emphasizes the importance of quantifying software products, processes and resources in order to manage software development successfully. In recent years, many software measures have been introduced in order to gather information about these aspects of software development. Among them, software size captures one of the most significant internal attribute of a software product. Over the years, it has been used for software assessment, estimation and improvement [2]-[5]. Although other factors such as software type, operational domain [6]-[8], developers' skills [9], [10] and tools and methods used [11], [12] have a considerable influence, software size remains the dominant factor with respect to the effort of software development [13]-[16].

Among the software size measures, Function Point (FP) [17] has achieved a wide acceptance in sizing software products [18]-[20]. Although its applicability is limited to procedural business systems, many researchers agree that the main idea of FP can be extended in order to be successfully used it for other types of systems [18], [21]-[24]. As a result, a number of extensions to FP specialized for specific types of systems, ranging from real-time and embedded systems [25], [26] to Object-Oriented (OO) [24], [27]-[29] and web-based [30]-[32] systems have been proposed.

In [33], we proposed an FP like measure, named Component Point (CP), for measuring the system-level

size of a Component-Based System (CBS) specified in Unified Modeling Language (UML) [34]. The CP measure is a promising approach to sizing a CBS. However, any proposed software measure becomes acceptable only if its validity has been proven via a thorough validation process. In fact, such a rigorous validation process requires two kinds of validations, namely a theoretical and an empirical validation. A theoretical validation confirms that the proposed measure satisfies all the necessary conditions that characterize the concept that it is supposed to measure, whereas an empirical validation verifies its usefulness against some external attributes [24]. In this paper, we provide a theoretical validation for the CP measure using mathematics in order to prove that not only the CP measure holds all the mathematical conditions necessary for a size measure but it is also useful in predicting the effort of Component-Based Software Development (CBSD). Our aim is to show that the CP measure can be used in CBSD in a similar way that FP and its extensions are used in other software development paradigms. We refer any interested readers to [33] for: (i) a review of the related work; (ii) the rationale behind the proposal of the CP measure; and (iii) a detailed explanation of the CP counting process.

2. Component Point Measure

The CP measure [33] integrates three existing software measures, namely the Class Point [24] size measure, which is an extension to FP for OO systems and two complexity measures [35] for CBS. It then quantifies components in a CBS in analogy to the class point counting performed for the Class Point estimate. The rationale behind this is to extend an existing measure from the more matured OO paradigm to the related and relatively young CBS discipline.

The CP counting process consists of four main steps [33]. In the first step, the UML specification is analyzed in order to identify the types of the components in a CBS. The identified components are then classified into three types, namely User Components (UC), Service Components (SC) and Domain Components (DC) as suggested in [36].

In the second step, the Interface Complexity (IFC) and Interaction Complexity (ITC) of each component are evaluated. For this purpose, we use the two complexity measures for CBS described in [35].

$$IFC = \sum_{j=1}^{2} \sum_{k=1}^{3} I_{jk} \times W_{jk} \tag{1}$$

where I_{jk} is the number of interfaces of type j (Internal Logical File or External Interface File) with complexity level k (Low, Average or High) and W_{jk} is the weight given for the interface type j with complexity level k. The complexity level of each interface is evaluated based on the Number of Operations (NO) and Number of Parameters (NP) derived from the operation signatures for each interface [35].

$$ITC = \sum_{i=1}^{l} \sum_{j=1}^{m} (IF_{ij} \times \sum_{k=1}^{n} CM_{ijk}) \tag{2}$$

where l is the number of interfaces that the component has, m is number of operations that the i interface provides, IF_{ij} is the interaction frequency of the j operation of the i interface, n is the number of data types involved in the information content of the interaction performed by the j operation of the i interface and finally, CM_{ijk} is the complexity measure of the k data type of the information content involved for the interaction performed by the j operation of the i interface. The Interaction Frequency (IF) for an operation of a given interface is defined as the ratio of the number of interactions (N_O) performed by the operation and the number of interactions (N_I) performed by all operations of the interface; whereas the Complexity

Measure (*CM*) of a data type of the information content involved for an interaction is defined as the sum of complexities of all its member data types derived from the signature of the corresponding operation [35]. Based on both count and complexity of the interfaces and interactions, the complexity of each component is then evaluated.

In the third step, the components are weighted based on their type and complexity and the Unadjusted Component Point (UCP) count is computed as a weighted sum.

$$UCP = \sum_{i=1}^{3}\sum_{j=1}^{3} C_{ij} \times W_{ij} \tag{3}$$

where C_{ij} is the number of components of type i (UC, SC or DC) with complexity level j (Low, Average or High) and W_{ij} is the weight given for the component type i with complexity level j.

In the fourth step, the UCP count is adjusted with an optional Value Adjustment Factor (VAF) obtained by considering the degree of influence of 14 General System Characteristics (GSC) of the CBS. The summation of the influence degrees of the 14 GSC gives the Total Degree of Influence (TDI), which is then used to compute the VAF.

$$VAF = 0.65 + (0.01 \times TDI) \tag{4}$$

The UCP count is then adjusted with the VAF and the final CP count is computed.

$$CP = UCP \times VAF \tag{5}$$

Further information on the review of the related work, the rationale behind the proposal of the CP measure, a detailed explanation of the CP counting process and an application of the CP measure to a real-world CBS can be found in [33].

3. Validation of CP Measure

The CP measure is a promising approach for sizing a CBS. However, any proposed software measure becomes acceptable only if its validity has been proven via a thorough validation process. In fact, such a rigorous validation process requires two kinds of validations, namely a theoretical and an empirical validation. A theoretical validation confirms that the proposed measure satisfies all the necessary conditions that characterize the concept that it is supposed to measure, whereas an empirical validation verifies its usefulness against some external attributes [24]. However, due to two reasons, we see that the theoretical validation is an essential prerequisite before an empirical validation takes place. First, any exploratory process of looking for correlations is not an acceptable scientific method of validation in itself if it is not accompanied by a sound theoretical foundation to support it [37]. Second, once the theoretical soundness of any proposed measure has been proven, it is more likely that it can be used to build models that correlate the internal software attributes to external ones, since the hypotheses behind these models are often expressed as the correlations between the internal and external attributes. Thus, in this section, we provide a theoretical validation for the CP measure.

There have been several theoretical underpinnings [38]-[41] proposed for the formal validation of software metrics. Among them, we use the mathematical framework proposed by Briand et al. [41] for the theoretical validation of software metrics. There are three reasons for the selection of this framework for the purpose. First, it is a generic framework because it is not specific to any particular software artifact;

second, it is more rigorous as it is based on precise mathematical concepts; and third, it has been successfully used for the theoretical validation of the Class Point size measure [24].

The CP measure has been composed of three existing software measures, namely the Class Point size measure for OO systems [24] and two complexity measures for a CBS [35]. Therefore, when theoretically validating the CP measure, it is not enough to prove that the CP measure is valid against the necessary mathematical conditions that it should hold but also to confirm that both Class Point size measure and the two component complexity measures on which the CP measure was formulated satisfy their corresponding necessary mathematical conditions with respect to the same theoretical framework. In [24], the Class Point measure has been validated against Briand et al.'s mathematical framework. However, such a formal validation is not available for the component interface and interaction complexity measures proposed in [35]. Therefore, we first validate the component interface and interaction complexity measures against Briand et al.'s mathematical framework. Then we validate the CP measure against the same mathematical framework and prove that the CP size measure holds all the necessary mathematical conditions specific for a size measure.

4. Theoretical Validation

In [41], Briand *et al.* have proposed a mathematical framework for the theoretical validation of software metrics. According to *this* framework, any software system (S) can be characterized by its elements (E) and the relations (R) among them. Thus, if S stands for a CBS, it can also be represented as a pair of $<E,R>$ where E represents the components of S and R is a relation on $E(R \subseteq E \times E)$ representing the interactions among its components. Given a system $S =< E,R >$, a module $m =< E_m, R_m >$ becomes a sub-system of S if and only if $E_m \subseteq E$, $R_m \subseteq E_m \times E_m$ and $R_m \subseteq R$.

The framework defines complexity as a system-level property which depends on its relationships. However, the authors themselves have noted that the same mathematical framework can be used to define the complexity of individual elements of a system if the relations among their sub-elements are considered [41]. A component of a CBS can be characterized by its interface and interactions. Thus, according to the above mathematical framework, if S stands for a component of a CBS, it can also be represented as a pair of $<E,R>$ where E and R represent its interface and interactions, respectively. Given a component $S =< E,R >$, a set of its interfaces $m =< E_m, R_m >$ becomes a sub-component of S if and only if $E_m \subseteq E$, $R_m \subseteq E_m \times E_m$ and $R_m \subseteq R$.

4.1. Interface and Interaction Complexity Measures

The CP approach uses an Interface Complexity (IFC) measure and an Interaction Complexity (ITC) measure [35] to evaluate the complexity of components in a CBS. The mathematical framework requires a complexity measure to satisfy five necessary conditions for its formal validation [41]. In particular, a software complexity measure should,

1) Hold non-negativity. That is if $S =< E,R >$, then $Complexity(S) \geq 0$.
2) Be null when there is no relationship among the elements of the system. That is if $S =< E,R >$ and $R = \emptyset$, then $Complexity(S) = 0$.
3) Satisfy symmetry. This means that if $S =< E,R >$ and $S^{-1} =< E,R^{-1} >$, then $Complexity(S) = Complexity(S^{-1})$.
4) Hold module monotonicity. That is if $S =< E,R >$, $m =< E_m, R_m >$, $n =< E_n, R_n >$, $m \cup n \subseteq S$ and $R_m \cap R_n = \emptyset$, then $Complexity(S) \geq Complexity(m) + Complexity(n)$.

5) Hold disjoint module additivity. This means that if $S =< E, R >$, $m =< E_m, R_m >$, $n =< E_n, R_n >$, $S = m \cup n$ and $m \cap n = \emptyset$, then $Complexity(S) = Complexity(m) + Complexity(n)$.

The validation of IFC and ITC measures of a component against Briand *et al.'s* mathematical framework is given below.

Condition **1:** The IFC and ITC of component $S =< E, R >$ should be non-negative. That is $IFC(S) \geq 0$ and $ITC(S) \geq 0$.

Proof: From (1),

$IFC(S) = \sum_{j=1}^{2} \sum_{k=1}^{3} I_{jk} \times W_{jk}$ where $I_{jk} \geq 0$ and $W_{jk} \geq 0$ for all *j* and *k*.

Thus, $IFC(S) \geq 0$

The Interface Complexity (IFC) measure of component holds non-negativity.

From (2),

$ITC(S) = \sum_{i=1}^{l} \sum_{j=1}^{m} (IF_{ij} \times \sum_{k=1}^{n} CM_{ijk})$ where $IF_{ij} \geq 0$ and $CM_{ijk} \geq 0$ for all *i*, *j* and *k*.

Thus, $ITC(S) \geq 0$

The Interaction Complexity (ITC) measure of component holds non-negativity.

Condition 2: The IFC and ITC of component $S =< E, R >$ should be null if R is empty. That is $R = ? \Rightarrow IFC(S) =$ and $R = ? \Rightarrow ITC(S) =$.

Proof: If $R = \emptyset$, it implies that there is no interaction performed by the interfaces of component *S*.

From (1),

$IFC(S) = \sum_{j=1}^{2} \sum_{k=1}^{3} I_{jk} \times W_{jk}$ where I_{jk} is the number of interfaces of type *j* with the complexity level *k* in component *S*.

When $j = 1$ and $j = 2$, I_{jk} stands for number of interfaces classified into Internal Logical File (ILF) and External Interface File (EIF) types, respectively. The interfaces that consist of operations and exchange data become candidates for this classification [35].

Thus, $R = ? \Rightarrow I_{jk} =$

$R = ? \Rightarrow IFC(S) =$

The Interface Complexity (IFC) measure of component satisfies the null value condition.

From (2),

$ITC(S) = \sum_{i=1}^{l} \sum_{j=1}^{m} (IF_{ij} \times \sum_{k=1}^{n} CM_{ijk})$ where IF_{ij} is the interaction frequency of the *j* operation of the *i* interface in component *S*.

$R = ? \Rightarrow IF_{ij} =$

Thus, $R = ? \Rightarrow ITC(S) =$

The Interaction Complexity (ITC) measure of component satisfies the null value condition.

Condition 3: The IFC and ITC of component $S =< E, R >$ should hold symmetry. That is, if $S =< E, R >$ and $S^{-1} =< E, R^{-1} >$, then $IFC(S) = IFC(S^{-1})$ and $ITC(S) = ITC(S^{-1})$.

Proof: Since the directions of the interactions performed are not considered when evaluating the interface and interaction complexities of a component, both the Interface Complexity (IFC) and the Interaction Complexity (ITC) measures of a component hold this condition.

Condition 4: The IFC and ITC of component $S =< E, R >$ should hold module monotonicity. That is if $S =< E, R >, m =< E_m, R_m >, n =< E_n, R_n >, m \cup n \subseteq S$ and $R_m \cap R_n = \emptyset$, then $IFC(S) \geq IFC(m) + IFC(n)$ and $ITC(S) \geq ITC(m) + ITC(n)$.

Proof: Let *m* and *n* be two sub-components of component *S* such that $m \cup n \subseteq S$ and $R_m \cap R_n = \emptyset$.

From (1), the sum of interface complexities of two sub-components *m* and *n* can be expressed as,

$IFC(m) + IFC(n) = \sum_{j=1}^{2} \sum_{k=1}^{3} \left[(I_m)_{jk} + (I_n)_{jk} \right] \times W_{jk}$ where $(I_m)_{jk}$ and $(I_n)_{jk}$ are the number of interfaces of type *j* with complexity level *k* in sub-components *m* and *n*, respectively.

Since $m \cup n \subseteq S$ and $R_m \cap R_n = \emptyset$, $(I_m)_{jk} + (I_n)_{jk} \leq I_{jk}$ for all *j* and *k* where I_{jk} is the number of interfaces of type *j* with complexity level *k* in component *S*.

Thus, $IFC(m) + IFC(n) \leq \sum_{j=1}^{2} \sum_{k=1}^{3} I_{jk} \times W_{jk}$

$IFC(S) \geq IFC(m) + IFC(n)$

The Interface Complexity (IFC) measure of component satisfies the condition of module monotonicity.

From (2), the sum of interaction complexities of two sub-components *m* and *n* can be expressed as,

$ITC(m) + ITC(n) = \sum_{i=1}^{a+b} \sum_{j=1}^{m} (IF_{ij} \times \sum_{k=1}^{n} CM_{ijk})$ where *a* and *b* are the number of interfaces in sub-components *m* and *n*, respectively.

Since $m \cup n \subseteq S$ and $R_m \cap R_n = \emptyset$, $a + b \leq l$ where *l* is the number of interfaces in component *S*.

Thus, $ITC(m) + ITC(n) \leq \sum_{i=1}^{l} \sum_{j=1}^{m} (IF_{ij} \times \sum_{k=1}^{n} CM_{ijk})$

$ITC(S) \geq ITC(m) + ITC(n)$

The Interaction Complexity (ITC) measure of component satisfies the condition of module monotonicity.

Condition 5: The IFC and ITC of component $S =< E, R >$ should hold disjoint module additivity. That is if $S =< E, R >, S = m \cup n$ and $m \cap n = \emptyset$, then $IFC(S) = IFC(m) + IFC(n)$ and $ITC(S) = ITC(m) + ITC(n)$.

Proof: Let *m* and *n* be two sub-components of component *S* such that $S = m \cup n$ and $m \cap n = \emptyset$.

From (1), the sum of interface complexities of two sub-components *m* and *n* can be expressed as,

$IFC(m) + IFC(n) = \sum_{j=1}^{2} \sum_{k=1}^{3} \left[(I_m)_{jk} + (I_n)_{jk} \right] \times W_{jk}$ where $(I_m)_{jk}$ and $(I_n)_{jk}$ are the number of interfaces of type *j* with complexity level *k* in sub-components *m* and *n*, respectively.

Since $S = m \cup n$ and $m \cap n = \emptyset$, $(I_m)_{jk} + (I_n)_{jk} = I_{jk}$ for all *j* and *k* where I_{jk} is the number of interfaces of type *j* with complexity level *k* in component *S*.

Thus, $IFC(m) + IFC(n) = \sum_{j=1}^{2} \sum_{k=1}^{3} I_{jk} \times W_{jk}$

$IFC(S) = IFC(m) + IFC(n)$

The Interface Complexity (IFC) measure of component satisfies the condition of disjoint module additivity.

From (2), the sum of interaction complexities of two sub-components *m* and *n* can be expressed as,

$ITC(m) + ITC(n) = \sum_{i=1}^{a+b} \sum_{j=1}^{m} (IF_{ij} \times \sum_{k=1}^{n} CM_{ijk})$ where *a* and *b* are the number of interfaces in sub-components *m* and *n*, respectively.

Since $S = m \cup n$ and $m \cap n = \emptyset$, $a + b = l$ where l is the number of interfaces in component S.

Thus, $ITC(m) + ITC(n) = \sum_{i=1}^{l}\sum_{j=1}^{m}(IF_{ij} \times \sum_{k=1}^{n} CM_{ijk})$

$$ITC(S) = ITC(m) + ITC(n)$$

The Interaction Complexity (ITC) measure of component satisfies the condition of disjoint module additivity.

The above proofs show that the IFC and ITC measures are valid against Briand et al.'s mathematical framework [41] for the theoretical validation of software metrics. In particular, we have proven that both the IFC and ITC measures hold non-negativity, are null if there are no interactions among the interfaces of the components, satisfy symmetry, hold the condition of module monotonicity and satisfy the condition of disjoint module additivity. Therefore, both Class Point size measure [24] and two component complexity measures [35] on which the CP measure was formulated satisfy their corresponding necessary mathematical conditions with respect to the same theoretical framework.

4.2. CP Size Measure

The mathematical framework requires a software size measure to satisfy six necessary conditions for its formal validation [41]. In particular, a software size measure should,

1. hold non-negativity. That is if $S =< E, R >$, then $Size(S) \geq 0$.
2. be null when there is no element in the system. This means that if $S =< E, R >$ and $E = \emptyset$, then $Size(S) = 0$.
3. hold module additivity. That is if $S =< E, R >$, $m =< E_m, R_m >$, $n =< E_n, R_n >$, $m \subseteq S$, $n \subseteq S$, $E = E_m \cup E_n$ and $E_m \cap E_n = \emptyset$, then $Size(S) = Size(m) + Size(n)$.
4. be given by the knowledge of the sizes of all disjoint elements of the system. That is if $S =< E, R >$, then $Size(S) = \sum_{e \in E} Size(m_e)$.
5. hold monotonicity. This means that if $S =< E, R >$, $m =< E_m, R_m >$ and $E_m \subseteq E$, then $Size(m) \leq Size(S)$.
6. be never greater than the sum of the sizes of any pair of sub-systems of the system. This means that, if $S =< E, R >$, $m =< E_m, R_m >$, $n =< E_n, R_n >$, $m \subseteq S$, $n \subseteq S$ and $E = E_m \cup E_n$, then $Size(S) \leq Size(m) + Size(n)$.

The validation of CP measure against Briand et al.'s mathematical framework is given below.

Condition 1: The CP measure of system $S =< E, R >$ should be non-negative. That is $CP(S) \geq 0$.

Proof: From (3),

$UCP(S) = \sum_{i=1}^{3}\sum_{j=1}^{3} C_{ij} \times W_{ij}$ where $C_{ij} \geq 0$ and $W_{ij} \geq 0$ for all *i* and *j*.

Thus, $UCP(S) \geq 0$

From (4) and (5),

$VAF = 0.65 + (0.01 \times TDI)$ and $CP(S) = UCP(S) \times VAF$ where $VAF \geq 0$ for all *TDI*.

Thus, $CP(S) \geq 0$

The CP measure holds non-negativity.

Condition 2: The CP measure of system $S =< E, R >$ should be null if E is empty. That is

$E = ? \Rightarrow CP(S) =$.

Proof: If $E = \emptyset$, it implies that there is no component present in system *S*.

From (3),

$UCP(S) = \sum_{i=1}^{3}\sum_{j=1}^{3} C_{ij} \times W_{ij}$ where C_{ij} is number of components of type *i* with complexity level *j* in system *S*.

Thus, $E = ? \Rightarrow C_{ij} =$

$E = ? \Rightarrow UCP\ S) = 0$

From (5),

$CP(S) = UCP(S) \times VAF$

Thus, $E = ? \Rightarrow CP\ S) = 0$

The CP measure satisfies null value condition.

Condition 3: The CP measure of system $S =< E, R >$ should hold module additivity. That is if $m \subseteq S$, $n \subseteq S$, $E = E_m \cup E_n$ and $E_m \cap E_n = \emptyset$, then $CP(S) = CP(m) + CP(n)$.

Proof: Let *m* and *n* be two sub-systems of *S* such that $m \subseteq S$, $n \subseteq S$, $E = E_m \cup E_n$ and $E_m \cap E_n = \emptyset$.

From (3), the sum of the UCP counts of two sub-systems *m* and *n* can be expressed as,

$UCP(m) + UCP(n) = \sum_{i=1}^{3}\sum_{j=1}^{3}\left[(C_m)_{ij} + (C_n)_{ij}\right] \times W_{ij}$ where $(C_m)_{ij}$ and $(C_n)_{ij}$ are the number of components of type *i* with complexity level *j* in sub-systems *m* and *n*, respectively.

$E = E_m \cup E_n$ and $E_m \cap E_n = \emptyset$ imply that no modification is made to the components of CBS when the system is partitioned into sub-systems *m* and *n*. This means that for each component of *m* and *n* sub-systems, the values of IFC and ITC will be unchanged after the partitioning. Thus, $(C_m)_{ij} + (C_n)_{ij} = C_{ij}$ is for all *i* and *j* where C_{ij} is the number of components of type *i* with complexity level *j* in system *S*.

Thus, $UCP(m) + UCP(n) = \sum_{i=1}^{3}\sum_{j=1}^{3} C_{ij} \times W_{ij}$

$UCP(m) + UCP(n) = UCP(S)$

From (5),

$CP(S) = UCP(S) \times VAF$

Thus, $CP(S) = UCP(m) \times VAF + UCP(n) \times VAF$

$CP(S) = CP(m) + CP(n)$

The CP measure satisfies module additivity.

Condition 4: The CP measure of system $S =< E, R >$ should be given by the knowledge of the sizes of all disjoint sub-systems of *S*. That is if $S =< E, R >$, then $CP(S) = \sum_{e \in E} CP(m_e)$.

Proof: Let *m* and *n* be two sub-systems of *S* such that $S =< E, R >$, $m =< E_m, R_m >$, $n =< E_n, R_n >$, $m \subseteq S$, $n \subseteq S$, $E = E_m \cup E_n$ and $E_m \cap E_n = \emptyset$.

Since the CP measure satisfies Condition 3 above, $CP(S) = CP(m) + CP(n)$ for a system composed of two disjoint sub-systems.

If the system *S* is composed of *k* number of disjoint sub-systems,

Then, $CP(S) = \sum_{i=1}^{k} CP(m_i)$

If the sub-system *m* can be partitioned into disjoint components $m_e =< \{e\}, R_e >$,

Then, $CP(m) = \sum CP(m_e)$

Since $\bigcap E_m = \emptyset$ and $\bigcup E_m = E$ for all *m*,

$CP(S) = \sum_{e \in E} CP(m_e)$

The CP measure is given by the knowledge of the CP counts of all disjoint components of *S.*

Condition 5: The CP measure of system $S =< E, R >$ should hold monotonicity. That is if $S =< E, R >$, $m =< E_m, R_m >$ and $E_m \subseteq E$, then $CP(m) \le CP(S)$.

Proof: Let *m* and *n* be two sub-systems of *S* such that $m =< E_m, R_m >$, $n =< E_n, R_n >$, $E_m \subseteq E$ and $E_n \subseteq E$.

Since the CP measure satisfies Condition 3 above, $CP(S) = CP(m) + CP(n)$

Since the CP measure also holds Condition 1 above, $CP(S) \ge 0$, $CP(m) \ge 0$ and $CP(n) \ge 0$.

Thus, $CP(m) \le CP(S)$

The CP measure satisfies monotonicity condition.

Condition 6: The CP measure of system $S =< E, R >$ should not greater than the sum of the CP counts of any pair of its sub-systems. That is if $S =< E, R >$, $m =< E_m, R_m >$, $n =< E_n, R_n >$, $m \subseteq S$, $n \subseteq S$ and $E = E_m \cup E_n$, then $CP(S) \le CP(m) + CP(n)$.

Proof: Let *m* and *n* be two sub-systems of S such that $S =< E, R >$, $m =< E_m, R_m >$, $n =< E_n, R_n >$, $m \subseteq S$, $n \subseteq S$ and $E = E_m \cup E_n$.

From (3), the sum of the UCP counts of two sub-systems *m* and *n* can be expressed as,

$UCP(m) + UCP(n) = \sum_{i=1}^{3} \sum_{j=1}^{3} \left[(C_m)_{ij} + (C_n)_{ij} \right] \times W_{ij}$ where $(C_m)_{ij}$ and $(C_n)_{ij}$ are the number of components of type *i* with complexity level *j* of sub-systems *m* and *n*, respectively.

Since there can be presence of common components between the sub-systems *m* and *n*, $(C_m)_{ij} + (C_n)_{ij} \ge C_{ij}$ for all *i* and *j* where C_{ij} is the number of components of type *i* with complexity level *j* of system *S*.

Thus, $UCP(m) + UCP(n) \ge \sum_{i=1}^{3} \sum_{j=1}^{3} C_{ij} \times W_{ij}$

$UCP(m) + UCP(n) \ge UCP(S)$

From (5),

$CP(S) = UCP(S) \times VAF$

Thus, $CP(S) \le UCP(m) \times VAF + UCP(n) \times VAF$

$CP(S) \le CP(m) + CP(n)$

The CP measure of *S* is never greater than the sum of the CP counts of any pair of its sub-systems.

The above proofs show that the CP measure is valid against Briand et al.'s mathematical framework [41] for the theoretical validation of software metrics, given the fact that the CP measure holds non-negativity, becomes null if there are no components in a CBS, satisfies the condition of module additivity, is given by the knowledge of the sizes of all disjoint sub-systems of a CBS, holds monotonicity and never exceeds the sum of the sizes of any pair of sub-systems of a CBS.

5. Conclusions and Future Work

In this paper, we have provided a theoretical validation for the Component Point (CP) measure [33] which we have developed for measuring the system-level size of a CBS. This validation has proven that not only the CP measure is valid against all the necessary mathematical conditions that it should hold as a size measure, but also the three existing software measures on which the CP measure was formulated are valid against their corresponding necessary mathematical conditions with respect to the same theoretical framework. In our validation, we have proven the fact that the CP measure holds non-negativity, becomes null when there are no components in a CBS, satisfies the condition of module additivity, is given by the knowledge of the sizes of all disjoint sub-systems of a CBS, holds the condition of monotonicity, and never exceeds the sum of the sizes of any pair of sub-systems of a CBS.

Thus, we have shown that the CP measure not only holds all the mathematical conditions that are necessary for a size measure but also can be used in CBSD in a similar way that FP and its extensions are used in other software development paradigms. For future work, we intend to formulate an effort model that will use the CP count as well as other effort drivers of CBSD in order to obtain more realistic effort estimates for CBS.

References

[1] Standish Group. (2015). *Chaos Report 2015*. The Standish Group. Boston, Massachusetts, United States.

[2] Humphrey, W. S. (1989). *Managing the Software Process*. Boston, Massachusetts, United States: Addison-Wesley.

[3] Fenton, N. E. (1991). *Software Metrics: A Rigorous and Practical Approach*. London, United Kingdom: Chapman & Hall.

[4] Humphrey, W. S. (1995). *A Discipline for Software Engineering*. Boston, Massachusetts, United States: Addison-Wesley.

[5] Fenton, N. E., & Bieman, J. (2014). *Software Metrics: A Rigorous and Practical Approach* (3rd ed.). Boca Raton, Florida, United States: CRC Press.

[6] Boehm, B. W. (1981). *Software Engineering Economics*. Upper Saddle River, New Jersey, United States: Prentice Hall.

[7] DeMarco, T. (1984). An algorithm for sizing software products. *ACM SIGMETRICS Performance Evaluation Review*, *12(2)*, 13 – 22.

[8] Murali, C. S., & Sankar, C. S. (1997). Issues in estimating real-time data communications software projects. *Information and Software Technology*, *39(6)*, 399 – 402.

[9] Boehm, B. W., & Papaccio, P. N. (1988). Understanding and controlling software costs. *IEEE Transactions on Software Engineering*, *14(10)*, 1462 – 1477.

[10] Blackburn, J. D., Scudder, G. D., & Van Wassenhove, L. N. (1996). Improving speed and productivity of software development: a global survey of software developers. *IEEE Transactions on Software Engineering*, *22(12)*, 875 – 885.

[11] Banker, R. D., Kauffman, R. J., & Kumar, R. (1992). An empirical test of object-based output measurement metrics in a computer aided software engineering (CASE) environment. *Journal of Management Information Systems*, *8(3)*, 127 – 150.

[12] Chan, T., Chung, S. L., & Ho, T. H. (1996). An economic model to estimate software rewriting and replacement times. *IEEE Transactions on Software Engineering*, *22(8)*, 580 – 598.

[13] Verner, J. & Tate, G. (1992). A software size model. *IEEE Transactions on Software Engineering*, *18(4)*, 265 – 278.

[14] Cockcroft, S. K. S. (1996). Estimating CASE development size from outline specifications. *Information*

and Software Technology, *38(6)*, 391 – 399.
[15] Hakuta, M., Tone, F. & Ohminami, M. (1997). A software size estimation model and its evaluation. *Journal of Systems and Software*, *37(3)*, 253 – 263.
[16] MacDonell, S. G. (2003). Software source code sizing using fuzzy logic modelling. *Information and Software Technology*, *45(7*), 389 – 404.
[17] IFPUG. (2010). *Function Point Counting Practices Manual*, *Release 4.3.1*. International Function Point Users Group, Princeton, New Jersey, United States.
[18] Dreger, J. B. (1989). *Function Point Analysis*. Upper Saddle River, New Jersey, United States: Prentice-Hall.
[19] Kemerer, C. F., & Porter, B. S. (1992). Improving the reliability of function point measurement: An empirical study. *IEEE Transactions on Software Engineering*, *18(11)*, 1011 – 1024.
[20] Longstreet, D. H. (1995). How Are Function Points Useful?. *American Programmer*, *8(12)*, 25 – 32.
[21] Boehm, B. W., Clark, B., Horowitz, E., Westland, C., Madachy, R., & Selby, R. (1995). Cost models for future software life cycle processes: COCOMO 2.0. *Annals of Software Engineering*, *1(1)*, 57 – 94.
[22] Symons, C. R. (1988). Function Point analysis: Difficulties and improvements. *IEEE Transactions on Software Engineering*, *14(1)*, 2 – 11.
[23] Banker, R. D., Kauffman, R. J., Wright, C., & Zweig, D. (1994). Automating output size and reuse metrics in a repository-based computer-aided software engineering (CASE) environment. *IEEE Transactions on Software Engineering*, *20(3)*, 169 – 187.
[24] Costagliola, G., Ferrucci, F., Tortora, G., & Vitiello, G. (2005). Class point: An approach for the size estimation of object-oriented systems. *IEEE Transactions on Software Engineering*, *31(1)*, 52 – 74.
[25] Jones, C. (1986). *Programming productivity*, New York, United States: McGraw-Hill.
[26] Abran, A., Maya, M., Desharnais, J. M., & St-Pierre, D. (1997). Adapting function points to real-time software. *American Programmer*, *10(11)*, 32 – 43.
[27] Whitmire, S. A. (1996). 3D function points: Applications for object-oriented software. *Proceedings of the Applications in Software Measurements Conference*. San Diego, California, United States.
[28] Minkiewicz, A. F. (1997). Measuring Object Oriented Software with Predictive Object Points. *Proceedings of the Applications in Software Measurements Conference*. Atlanta, Georgia, United States.
[29] Antoniol, G., Lokan, C., Caldiera, G., & Fiutem, R. (1999). Function point-like measure for object-oriented software. *Empirical Software Engineering*, *4(3)*, 263 – 287.
[30] Reifer, D. (2000). Web-development: Estimating quick-time-to-market software. *IEEE Software*, *17(6)*, 57 – 64.
[31] Cleary, D. (2000). Web-Based Development and Functional Size Measurement. *Proceedings of the IFPUG Annual Conference*. San Diego, California, United States.
[32] Cost Xpert. (2002). *Estimating Internet Development*. Cost Xpert Group, Inc. San Diego, California, United States.
[33] Wijayasiriwardhane, T., & Lai, R. (2010). Component point: A system-level size measure for component-based software systems. *Journal of Systems and Software*, *83(12)*, 2456-2470.
[34] Cheesman, J., & Daniels, J. (2001). *UML Components: A Simple Process for Specifying Component Based Software*. Boston, Massachusetts, United States: Addison-Wesley.
[35] Mahmood, S. & Lai, R. (2008). A complexity measure for UML component-based system specification. *Software - Practice & Experience*, *38(2)*, 117-134.
[36] Williams, J. (2001). The business case for components. In G. T. Heineman, & W. T. Councill (Eds.), *Component-Based Software Engineering: Putting the Pieces Together*. Boston, Massachusetts, United States: Addison-Wesley.

[37] Courtney, R. E., & Gustafson, D. A. (1992). Shotgun correlations in software measures. *Software Engineering Journal*, *8(1)*, 5 – 13.

[38] Weyuker, E. J. (1988). Evaluating software complexity measures. *IEEE Transactions on Software Engineering*, *14(9)*, 1357 – 1365.

[39] Schneidewind, N. F. (1992). Methodology for validating software metrics. *IEEE Transactions on Software Engineering*, *18(5)*, 410 – 422.

[40] Kitchenham, B., Pfleeger, S. L., & Fenton, N. (1995). Towards a framework for software measurement validation. *IEEE Transactions on Software Engineering*, *21(12)*, 929 – 944.

[41] Briand, L. C., Morasca, S., & Basili, V. R. (1996). Property-based software engineering measurement. *IEEE Transactions on Software Engineering*, *22(1)*, 68 – 86.

Thareendhra Wijayasiriwardhane obtained his Ph.D. in software engineering from La Trobe University, Melbourne, Australia in 2010. He also received a M.Sc. in computer science from University of Colombo School of Computing, Sri Lanka in 2005 and a B.Sc. with a first class honours from University of Kelaniya, Sri Lanka in 2000.

Presently, he is a senior lecturer attached to the Faculty of Science, University of Kelaniya, Sri Lanka. Thareendhra counts over 18 years of experience as an academic in the field of computing in Sri Lanka and Australia. His research interests include component-based systems, software measurement and estimation and software testing.

Dr. Wijayasiriwardhane has won number of awards for his academic and research excellence including the Endeavour Postgraduate Award from Australian Government, and Vice-Chancellor's Award for the Most Outstanding Young Researcher from University of Kelaniya, Sri Lanka.

Richard Lai obtained his Ph.D. from La Trobe University, Melbourne, Australia in 1990. He also received an M.Eng. and a B.Eng. with honours from University New South Wales, Australia in 1982 and 1979, respectively.

Presently, he is an associate professor attached to the Department of Computer Science and Information Technology, La Trobe University, Australia. Prior to joining La Trobe University in 1989, Richard spent over 10 years in the computer and communications industry. He has authored one book and more than 55 journal papers. His research interests include software economics, measurement, reliability and testing, requirements engineering, component-based systems and global software development.

Prof. Lai was ranked as the world's number one scholar in systems and software engineering consecutively for four years (1999–2002), according to an annual survey published in the Journal of Systems and Software.