Object-Oriented Cognitive-Spatial Complexity Measures

Varun Gupta and Jitender Kumar Chhabra

Abstract—Software maintenance and mainly software comprehension pose the largest costs in the software lifecycle. In order to assess the cost of software comprehension, various complexity measures have been proposed in the literature. This paper proposes new cognitive-spatial complexity measures, which combine the impact of spatial as well as architectural aspect of the software to compute the software complexity. The spatial aspect of the software complexity is taken into account using the lexical distances (in number of lines of code) between different program elements and the architectural aspect of the software complexity is taken into consideration using the cognitive weights of control structures present in control flow of the program. The proposed measures are evaluated using standard axiomatic frameworks and then, the proposed measures are compared with the corresponding existing cognitive complexity measures as well as the spatial complexity measures for object-oriented software. This study establishes that the proposed measures are better indicators of the cognitive effort required for software comprehension than the other existing complexity measures for object-oriented software.

Keywords—cognitive complexity, software comprehension, software metrics, spatial complexity, Object-oriented software

I. INTRODUCTION

SOFTWARE comprehension may account for over one third of the lifetime cost of a software system [1] and cost of software comprehension is directly affected by complexity of the software. The software complexity has been measured by many researchers using various affecting attributes such as control flow paths [2], the volume of operands and operators [3], identifier density [4], cognitive complexity [5], [6], [7], [8] and spatial complexity [9], [10], [11], [12]. Spatial complexity measures account for the difficulty of reading the source code of a program for understanding, in terms of the lexical distance (measured in lines of code) that the maintainer is required to traverse to follow control and/or data dependencies as they build a mental model [9]. This type of complexity is based on the spatial distance between the definition and use of various program elements. Understanding of the use of a program element also requires knowledge of control flow in which the program element has been used [13]. Thus, the type of control structure in which the program elements are being used, should also be considered to measure effort required for program comprehension. Many researchers have already stressed the importance of considering the kind of control structure while computing the cognitive complexity [6], [8], [13]. This architectural aspect of the complexity of the software can easily be reflected in cognitive complexity with help of using weights of various types of Basic Control Structures (BCS). Wang and Shao [6] identified the cognitive weights for various control structures. The measures of cognitive complexity proposed by various authors [6], [7], [8] considered only these weights, which were reflection of architectural viewpoint only and did not look into the spatial aspect at all. On the other hand, the importance of spatial distance towards complexity is well established and reported in [14], [9], [10], [11]. Thus it is very pertinent to combine the impact of architectural as well as spatial aspects of the software to compute the cognitive complexity. We will term this type of software complexity as cognitive-spatial complexity of the software.

The remainder of the paper is structured as follows: Section 2 presents the definitions of the proposed cognitive-spatial complexity metrics. Section 3 evaluates the proposed measures using Weyuker’s properties [15] and Section 4 validates the proposed measures using the Briand et al. framework [16]. Section 5 compares the proposed measures with the existing cognitive and spatial complexity measures for object-oriented software. Finally, Section 6 concludes the work.

II. PROPOSED COGNITIVE-SPATIAL COMPLEXITY MEASURES

In this paper, we have proposed two categories of measures of cognitive-spatial complexity of object-oriented software—class cognitive-spatial complexity (CSC), and object cognitive-spatial complexity (OCS). The class cognitive-spatial complexity (CCSC) measures the cognitive-spatial complexity of both members of the classes—methods and attributes. To understand the behaviour of any class, one needs to comprehend both the entities. The method’s code helps in understanding the processing logic and the attributes help in recognizing the properties of the class. The second category of proposed cognitive-spatial complexity is based on the definition of objects and usages of object-members. The cognitive-spatial complexity of object-oriented software is the combination of class cognitive-spatial complexity and object cognitive-spatial complexity.
A. Class cognitive-spatial complexity

In object-oriented software, class encapsulates the state and behavior of the concept it represents. It encapsulates state through attributes (or member/instance variables) and behavior through methods (member functions). So the effort needed to understand a class depends on the attributes and the methods of the class. Thus, class cognitive-spatial complexity consists of two parts – attribute cognitive-spatial complexity and method cognitive-spatial complexity.

1) Attribute cognitive-spatial complexity

The spatial complexity of an attribute is measured using the distance (in LOC) between its definition and first use within the method and subsequently taking into account the distance between two successive uses within the same method [11]. Since, attributes of a class represent data of the class. Understanding the use of data also requires knowledge of control flow in which the data has been used [13]. Thus, the type of control structure in which the attributes are used, should also be considered to measure program comprehension. If a data member is used in a simple assignment statement, understanding its purpose is much easier than the use of same member in a control statement. This aspect of the complexity of use of attributes can easily be reflected with help of using weights of various types of Basic Control Structures (BCS) as defined by Wang and Shao[6]. The cognitive-spatial complexity of an attribute at a particular point of use is the product of the cognitive weight of the BCS, in which the attribute is being used and the absolute difference (in LOC) of the current use of the attribute from its previous use/definition. Thus, Attribute Cognitive-Spatial Complexity of an attribute i at line number k is defined as: -

\[
ACSC(i,k) = W_k \times Distance(i,k)
\]

where \(W_k\) is the cognitive weight of the BCS, in which the attribute, i has been used at line number k and \(Distance(i,k)\) is the absolute difference (in LOC) of the current use of the attribute from its previous use/definition. In case of multiple files coming into picture for measurement of this distance, the distance is defined as:

Distance = (distance of first use of the attribute from the top of the current file) + (distance of definition of the attribute from the top of the file containing definition)

Thus, Attribute Cognitive-Spatial Complexity of an attribute i in a class is defined as the average of cognitive-spatial complexities of all uses of the attribute i.e.

\[
ACSC_i = \frac{\sum_{k=1}^{p} ACSC(i,k)}{p}
\]

where \(p\) represents the count of uses of the attribute i in the class.

Class Attribute Cognitive-Spatial Complexity of a class (CACSC) is defined as the average of Attribute Cognitive-Spatial Complexity (ACSCI) of all attributes of the class.

\[
CACSC = \frac{\sum_{i=1}^{q} ACSC_i}{q}
\]

where \(q\) is the total number of attributes in the class.

2) Method cognitive-spatial complexity

The spatial complexity of a method is defined in terms of distance (in LOC) from its definition to its direct call (without using objects) in the other methods of the same class [12]. More the distance between definition of the method and use/call of the method, more cognitive effort would be required to correlate its usages with its definition. The type of control structure in which the method is being called, would also affect the cognitive effort required for comprehension. Thus, the cognitive-spatial complexity of a method at a particular point of use is the product of the cognitive weight of the BCS, in which the method is being called and the absolute difference (in LOC) of the current call of the method from its previous call/definition. Thus, Method Cognitive-Spatial Complexity of a method i at line number k is defined as:

\[
MCSC(i,k) = W_k \times Distance(i,k)
\]

where \(W_k\) is the cognitive weight of the BCS, in which the method, i has been called at line number k and \(Distance(i,k)\) is the absolute difference (in LOC) of the current call/use of the method from its definition. In case of multiple files, the distance is defined as:

\[
Distance = (distance of call of the method from the top of the current file) + (distance of definition of the method from the top of the file containing definition)
\]

Thus, Method Cognitive-Spatial Complexity of a method i in a class is defined as the average of cognitive-spatial complexities of all calls of the method i.e.

\[
MCSC_i = \frac{\sum_{k=1}^{m} MCSC(i,k)}{m}
\]

where \(m\) is the total number of calls of the method i in the class.

Class Method Cognitive-Spatial Complexity of a class (CMCSC) is defined as the average of Method Cognitive-Spatial complexity (MCSC) of all methods of the class.

\[
CMCSC = \frac{\sum_{i=1}^{n} MCSC_i}{n}
\]

where \(n\) is the count of method in the class.

A class consists of attributes and methods; the class cognitive-spatial complexity is the summation of the class attribute cognitive-spatial complexity and the class method cognitive-spatial complexity i.e.

\[
CCSC = CACSC + CMCSC
\]

B. Object cognitive-spatial complexity measures

Objects are instances of the classes. Classes do not execute directly, but their instances are used in form of the objects in object-oriented software. The proposed object cognitive-spatial complexity estimates the cognitive effort needed to correlate various definitions of the objects with their
corresponding classes, and various usages of object-members to their respective definitions. Thus, object cognitive-spatial complexity is of two types—object definition cognitive-spatial complexity and object-member usage cognitive-spatial complexity.

1) Object definition cognitive-spatial complexity

The object definition spatial complexity (ODSC) of an object is defined as the distance of the definition of the object from the corresponding class declaration [11]. The type of statement in which the object is defined, would also affect the cognitive effort required for comprehension. Thus, the object definition cognitive-spatial complexity (ODCSC) of an object is the product of the cognitive weight of the BCS, in which the object is being defined and the absolute difference (in LOC) of the definition of the object from its class declaration. Thus, Object Definition Cognitive-Spatial Complexity (ODCSC) of an object \(i\) at line number \(k\) is defined as:

\[
ODCSC(i) = W_k \times Distance(i, k)
\]

where \(W_k\) is the cognitive weight of the BCS, in which the object, \(i\) has been defined at line number \(k\) and \(Distance(i, k)\) is the absolute difference (in LOC) of the definition of the object from the corresponding class declaration. In case of multiple files, the distance is defined as:

\[
Distance = (\text{distance of object definition from top of current file}) + (\text{distance of declaration of the corresponding class from the top of the file containing class})
\]

2) Object member usage cognitive-spatial complexity

The Object member usage cognitive-spatial complexity (OMUSC) of a member through a particular object is defined as the average of distances (in LOC) between definitions of the member in the corresponding class and calls of that member through the object [11]. The type of control structure in which the object-member is called/used, would also affect the cognitive effort required for comprehension. Thus, the cognitive-spatial complexity of an object-member at a particular point of use is the product of the cognitive weight of the BCS, in which the object-member is being defined and the absolute difference (in LOC) of the current use of the object-member from its previous use/definition. Thus, Object Member Usage Cognitive-Spatial Complexity of an object \(i\) at line number \(k\) is defined as:

\[
OMUSC(i,k) = W_k \times Distance(i, k)
\]

where \(W_k\) is the cognitive weight of the BCS, in which the object-member, \(i\) has been used at line number \(k\) and \(Distance(i, k)\) is the absolute difference (in LOC) of the current use of the object-member from its definition in the corresponding class. In case of multiple files coming into picture for measurement of this distance, the distance is defined as:

\[
Distance = (\text{distance of call from the top of the file containing call}) + (\text{distance of definition of the member from the top of the file containing definition})
\]

Thus, Object Member Usage Cognitive-Spatial Complexity of an object-member \(i\) is defined as the average of cognitive-

spatial complexities of all usages of the object-member \(i\), i.e.

\[
OMUCSC_i = \frac{1}{m} \sum_{k=1}^{m} OMUSC(i,k)
\]

where \(m\) is the total number of uses of the object-member \(i\).

Object Member Cognitive-Spatial Complexity of an object is defined as the average of Object-Member Usage Cognitive-Spatial Complexity (OMUCSC\(_i\)) of all members of the object.

\[
OMCSC_i = \frac{1}{n} \sum_{i=1}^{n} OMSC_i
\]

where \(n\) is the count of object-members being called through that object. The Object Cognitive-Spatial Complexity of an object is defined as summation of the Object Definition Cognitive-Spatial Complexity (ODCSC) of the object and the Object-Member Cognitive-Spatial complexity (OMCSC) i.e.

\[
OCSC = ODCSC + OMCSC
\]

III. EVALUATION OF THE PROPOSED MEASURES USING WEYUKER’S PROPERTIES

The new proposed measures are acceptable only when a validation process has proved their usefulness. In this section, we evaluate the proposed object-oriented cognitive-spatial measures using well-known nine Weyuker’s properties [15]. Weyuker proposed a formal list of nine properties for evaluating software complexity metrics. Many well-known authors have used these properties for evaluating complexity measures [17], [18], [19], [20], [21], [22]. We have applied Weyuker’s nine properties to evaluate the object oriented cognitive-spatial complexity measures proposed in Section 2. While describing these properties, \(P\) denotes an object-oriented program/class and \(Q\) represents its cognitive-spatial complexity, which will always be a non-negative number.

Property 1: This property states that \((3P)\), \((3Q)\) such that \((P) \neq (Q)\)

This property states that a complexity measure must not be “too coarse” such that it rates all programs as equally complex. Two object-oriented programs \(P\) and \(Q\) can always differ in values of class cognitive-spatial complexity measures or values of object cognitive-spatial complexity measures, since the measures are defined in terms of distances (in LOC), which will have most of the times different values for two different programs. Thus, object-oriented cognitive-spatial measures satisfy Property 1.

Property 2: Let \(c\) be a non-negative number, and then there is only finite number of programs of complexity \(c\).

This property is a strengthening of Property 1, which requires that the complexity measure must not be too “coarse”. Since, the class cognitive-spatial complexity is the summation of the attribute cognitive-spatial complexity and the method cognitive-spatial complexity of the class i.e.

\[
CCSC = CACSC + CMSC
\]

Further, attribute cognitive-spatial complexity depends on number of attributes and distances in LOC between the
different uses of the attributes and class method cognitive-spatial complexity depends on number of methods and distances between calls and definitions of the methods. Also, Object Cognitive-Spatial Complexity of an object is defined as summation of the Object Definition Cognitive-Spatial Complexity (ODCSC) of the object and the Object-Member Cognitive-Spatial complexity (OMCSC) i.e.

\[ \text{ODCSC}(i) = W_k \ast \text{Distance}(i, k) \]

and similarly, the object member cognitive-spatial complexity (OMCSC) is defined in terms of the cognitive weights of the BCS, in which the object is being defined and the absolute difference (in LOC) of the definition of the object from its class declaration i.e.

\[ \text{OMCSC}(i) = |\text{def}(i)| - |\text{usage}(i)| \]

where \( |\text{def}(i)| \) is the number of lines of code in the definition of object \( i \) and \( |\text{usage}(i)| \) is the number of lines of code in the usage of object \( i \).

**Property 3:** There are distinct programs \( P \) and \( Q \) such that \( |P| = |Q| \).

This property states that a complexity measure must not be “too fine” i.e. any specific value of the metric should not only be given by a single program. This property requires that complexity of a program. This property requires that complexity of a program P and Q respectively and CCSCPQ be the class cognitive-spatial complexity of the concatenated program of P and Q. Then, as per the definition of the measures, the resultant class cognitive-spatial complexity of the combined program would approximately be the sum of class cognitive-spatial complexities of individual programs, if they were independent and if the code of program Q was just appended after the code of program P, without disturbing the individual distances of definition and usage of members of class i.e.

\[ \text{CCSCPQ} = \text{CCSCP} + \text{CCSCQ} \]

Similarly, if OCSCP and OCSCQ are the object cognitive-spatial complexities of independent programs P and Q respectively and OCSCPQ is the object cognitive-spatial complexity of the concatenated program of P and Q, then, according to the definition of the measures, the resultant object cognitive-spatial complexity of the combined program would approximately be the sum of object cognitive-spatial complexities of individual programs, if they were independent i.e.

\[ \text{OCSCPQ} = \text{OCSCP} + \text{OCSCQ} \]

If programs P and Q were not independent, then the common classes and objects may appear once in the concatenated program. In that case the class/object cognitive-spatial complexity of common portion will contribute once in the measures and independent portions of both P and Q will continue to have their original contribution towards CCSCP as well as OCSCP. In that situation,

\[ \text{CCSCPQ} = \text{CCSCP} + \text{CCSCQ} - \text{CCSCcommon-classes} \]

and

\[ \text{OCSCPQ} = \text{OCSCP} + \text{OCSCQ} - \text{OCSCcommon-objects} \]

It is obvious that CCSCc ommon-class can never be greater than either CCSCP or CCSCQ. So in both cases whether P and Q are independent or not, it is evident that CCSCP ≤ CCSCPQ and CCSCQ ≤ CCSCPQ. Similarly, CCSCP ≤ OCSCPQ and OCSCQ ≤ OCSCPQ.

Thus, the object-oriented cognitive-spatial complexity metrics satisfy the Property 5.

**Property 4:** \( \exists (P) \) such that \( P = Q \) and \( |P| \neq |Q| \).

This property states that there is no one-to-one correspondence between functionality and complexity that means two programs having same functionality can have different complexity. Since, object-oriented cognitive-spatial complexity measures depend on the implementation details of programs. Two object-oriented programs P and Q having same functionality but different implementations will have different values of cognitive-spatial complexity i.e.

CCSCP ≠ CCSCQ and OCSCP ≠ OCSCQ may hold true for any two object-oriented programs P and Q where \( P = Q \).

Thus, Property 4 holds for the object-oriented cognitive-spatial complexity measures.

**Property 5:** \( \forall (P) \) such that \( |P| \leq |P;Q| \) and \( |Q| \leq |P;Q| \)

Property 5 states the concept of monotonicity with respect to composition. This property requires that complexity of a concatenated program obtained from the concatenation of two programs can never be less than the complexity of either of the programs. Let CCSCP and CCSCQ be the class cognitive-spatial complexity of object-oriented programs P and Q respectively and CCSCPQ be the class cognitive-spatial complexity of the concatenated program of P and Q. Then, as per the definition of the measures, the resultant class cognitive-spatial complexity of the combined program would approximately be the sum of class cognitive-spatial complexities of individual programs, if they were independent and if the code of program Q was just appended after the code of program P, without disturbing the individual distances of definition and usage of members of class i.e.

\[ \text{CCSCPQ} = \text{CCSCP} + \text{CCSCQ} \]

Similarly, if OCSCP and OCSCQ are the object cognitive-spatial complexities of independent programs P and Q respectively and OCSCPQ is the object cognitive-spatial complexity of the concatenated program of P and Q, then, according to the definition of the measures, the resultant object cognitive-spatial complexity of the combined program would approximately be the sum of object cognitive-spatial complexities of individual programs, if they were independent i.e.

\[ \text{OCSCPQ} = \text{OCSCP} + \text{OCSCQ} \]

If programs P and Q were not independent, then the common classes and objects may appear once in the concatenated program. In that case the class/object cognitive-spatial complexity of common portion will contribute once in the measures and independent portions of both P and Q will continue to have their original contribution towards CCSCP as well as OCSCP. In that situation,

\[ \text{CCSCPQ} = \text{CCSCP} + \text{CCSCQ} - \text{CCSCcommon-classes} \]

and

\[ \text{OCSCPQ} = \text{OCSCP} + \text{OCSCQ} - \text{OCSCcommon-objects} \]

It is obvious that CCSCcommon-class can never be greater than either CCSCP or CCSCQ. So in both cases whether P and Q are independent or not, it is evident that CCSCP ≤ CCSCPQ and CCSCQ ≤ CCSCPQ. Similarly, CCSCP ≤ OCSCPQ and OCSCQ ≤ OCSCPQ.

Thus, the object-oriented cognitive-spatial complexity metrics satisfy the Property 5.
complexities of programs obtained after concatenating program P with P, and R with Q respectively. Also, \((\exists P ) (\exists Q) (OCSCP = OCSCQ \& OCSPR \neq OCSCQR)\) and \((\exists P ) (\exists Q) (OCSCP = OCSCQ \& OCSPR \neq OCSCQR)\) where OCSCP, OCSCQ, OCSPR, OCSCQR, OCSPR and OCSCQR represent the object cognitive-spatial complexities of programs P, Q, P concatenated with R, Q concatenated with R, R concatenated with P, and R concatenated with Q respectively. Thus, property 6a and 6b are well satisfied by the object-oriented cognitive-spatial complexity measures.

Property 7: This property says that there are two programs P and Q such that Q is formed by permuting the order of the statements of P and \(|P| \neq |Q|\) that means a complexity measure should be sensitive to the permutation of statements.

The values of object-oriented cognitive-spatial complexity measures depend on the distances (in LOC) between uses of different program elements such as class-members and object-members. Thus, cognitive-spatial complexity of an object-oriented program depends on the order of statements of the program. When program Q is formed by permuting the order of the statements of the program P, then values of object-oriented cognitive-spatial complexity measures for the program Q will differ from the values of the measures obtained from the program P due to the change in LOC between different program elements. Thus, for programs P and Q, CCSCP \neq CCSCQ and OCSCP \neq OCSCQ where program Q is formed by permuting the order of the statements of program P. Hence, the object-oriented cognitive-spatial complexity measures satisfy property 7.

Property 8: If P is a renaming of Q, then \(|P| = |Q|\)

Since, names of a program and its elements do not contribute to the values of the object-oriented cognitive-spatial complexity measures, thus renaming of an object-oriented program does not have any effect on its cognitive-spatial complexity i.e. CCSCP = CCSCQ and OCSCP = OCSCQ where P is a renaming of Q. Thus, property 8 is well satisfied by the object-oriented cognitive-spatial complexity measures.

Property 9: \((\exists P ) (\exists Q)\) such that \((|P| + |Q| < |P; Q|)\)

According to property 9, the complexity of a new program obtained from the combinations of two programs, can be greater than the sum of complexities of two individual programs. This can be true for both the class cognitive-spatial complexity as well as the object cognitive-spatial complexity measures as classes and objects are brought into a single program from two different programs, this may increase distances (in LOC) between different program elements. For instance, the method cognitive-spatial complexity of a method of class depends on the distances (in LOC) between the calls and the definition of the method and this distance may increase in the combined program due to the presence of other program elements. Thus, same method when used in the combined program may contribute more to the class cognitive-spatial complexity than its use in the individual program. Similarly, it holds for attribute cognitive-spatial complexity also. Further, the object definition cognitive-spatial complexity (ODSC) of an object depends on the distance of the definition of the object from the corresponding class declaration and this distance may increase in the combined program due to introduction of new classes and objects in the new program. The same logic is also applicable in case of object-member cognitive-spatial complexity measures. Thus, the same object when used in the combined program may contribute more to the object cognitive-spatial complexity than its use in the individual program. Hence, the cognitive-spatial complexity of a combined program might be more than the sum of the cognitive-spatial complexities of individual programs i.e. \((\exists P ) (\exists Q)\) such that \((CCSCP + CCSCQ < CCSCPQ)\) where CCSCP, CCSCQ and CCSCPQ denote the class cognitive-spatial complexity of programs P, Q and P; Q respectively. Similarly, \((\exists P ) (\exists Q)\) such that \((OCSCP + OCSCQ < OCSCPQ)\) where OCSCP, OCSCQ and OCSCPQ represent object cognitive-spatial complexity of programs P, Q and P; Q respectively. Thus, object-oriented cognitive-spatial complexity measures satisfy property 9.

IV. APPLICATION OF BRIAND ET AL. FRAMEWORK

In this section, we validate object-oriented cognitive-spatial complexity measures by using the evaluation framework given by Briand et al. [16] to provide more credibility to the object-oriented cognitive-spatial complexity measures. In the framework, Briand et al. have given five properties, which a complexity measure must satisfy to be a useful complexity measure. Before the application of the framework, the basic definitions required for the evaluation process are given.

System: An object-oriented system S is represented as a \(<E, R>\), where E represents the set of elements of S, and R is a binary relation on E (R \subseteq E \times E) representing the relationships between elements of S. For the purpose of evaluation of object-oriented cognitive-spatial complexity measures, E is defined as the set of all classes, objects and their members in the program and R as the set of definitions of classes and objects and usages of classes and objects through their members.

Module: For a given object-oriented system S = \(<E, R>\), a system m = \(<Em, Rm>\) is a module of S if and only if Em \subseteq E, R \subseteq Em \times Em and Rm \subseteq R. A module m may be a class or a subprogram.

Complexity: The cognitive-spatial complexity of an object oriented system S is a function Complexity(S) that is described by Property 1 to Property 5.

Property 1 (Nonnegative): The complexity of a system S = \(<E, R>\) is nonnegative if Complexity (S) \geq 0.

Proof: Object-oriented cognitive-spatial complexity measures are defined in terms of weights of BCS and distances (in LOC), which are always non-negative numbers. Thus, cognitive-spatial complexity of an object-oriented system will always be non-negative. Hence, the proposed measures satisfy Property 1.

Property 2 (Null Value): The complexity of a system S = \(<E, R>\) is null if R is empty i.e. R = \(Ø\) \Rightarrow Complexity (S) = 0.
Proof: For object-oriented cognitive-spatial complexity measures, $R$ has been defined above as the set of definitions of all classes, objects, and their members and usages of classes and objects through their members. If there were no definitions of classes and objects or usages of class-members and object-members in the program, values of the class cognitive-spatial complexity (CCSC) measures as well as object cognitive-spatial complexity (OCSC) measures will be zero. Thus, property 2 is well satisfied by the proposed measures.

Property 3 (Symmetry): The complexity of a system $S = \langle E, R \rangle$ does not depend on the convention chosen to represent the relationships between its elements i.e. $(S = \langle E, R \rangle$ and $S^{-1} = \langle E, R^{-1} \rangle \Rightarrow \text{Complexity}(S) = \text{Complexity}(S^{-1})$).

Proof: According to this property, a complexity measure should not be sensitive to representation conventions used for relationships. A relationship can be represented by two equivalent representation conventions “active” ($R$) or “passive” ($R'$) form [16]. As defined in the beginning of this section, relationships for the object-oriented cognitive-spatial complexity measures represent the set of definitions of classes and objects and usages of classes and objects through their members. As per the definitions of the proposed measures, count of these relationships is taken into consideration during measurement, not the conventions used to represent these relationships. Thus, object-oriented cognitive-spatial complexity of a program does not depend on the convention chosen to represent these relationships. Hence, the proposed measures satisfy property 3 also.

Property 4 (Module Monotonicity): The complexity of a system $S = \langle E, R \rangle$ is no less than the sum of the complexities of any two of its modules with no relationships in common i.e. $(S = \langle E, R \rangle$ and $m_1 = \langle \text{Em}_1, \text{Rm}_1 \rangle$ and $m_2 = \langle \text{Em}_2, \text{Rm}_2 \rangle$ and $m_1 \bigcup m_2 \subseteq S$ and $\text{Rm}_1 \bigcap \text{Rm}_2 = \emptyset \Rightarrow \text{Complexity}(S) \geq \text{Complexity}(m_1) + \text{Complexity}(m_2)$).

Proof: This property is analogous to Weyuker’s property 9 and as already explained above, the object-oriented cognitive-spatial complexity measures satisfy this property of module monotonicity very well. Thus, property 4 also holds for the proposed measures.

Property 5 (Disjoint Module Additivity): The complexity of a system $S = \langle E, R \rangle$ composed of two disjoint modules $m_1$, $m_2$, is equal to the sum of the complexities of the two modules i.e. $(S = \langle E, R \rangle$ and $S = m_1 \bigcup m_2$, and $m_1 \cap m_2 = \emptyset \Rightarrow \text{Complexity}(S) = \text{Complexity}(m_1) + \text{Complexity}(m_2)$).

Proof: If two independent subprograms/classes are combined together in a single program, then there will be no usage of class/object or their members of one subprogram from the other subprogram as the two subprograms are totally independent. Thus, their individual cognitive-spatial complexities will not get affected due to the combination of the two. The cognitive-spatial complexity value of the combined program will definitely be the summation of cognitive-spatial complexity values of the subprograms. Thus, the proposed measures satisfy property 5 very well.

V. COMPARISON OF THE PROPOSED MEASURES WITH THE COGNITIVE AND SPATIAL MEASURES

The Cognitive Functional Size (CFS) [5] metric proposed by Wang et al. mainly measure algorithmic complexity of a program independent of the language implementation. The CFS metric is not able to indicate comprehension level of a program completely since the cognitive effort required to comprehend a program also depends on the programming language in which the particular program has been written as different languages have different levels and the language level has been reported to affect cognitive efforts of understanding the programs [3], [23], [24]. Thus, the cognitive measures should also take into account the implementation details of a program while measuring the effort required for comprehension of the program. Similarly, the spatial complexity measures [9], [10], [11], [12] are based on the spatial distance between the definition and use of various program elements. However, many researchers have already stressed the importance of considering the program control flow [13] and the kind of control structure while computing the cognitive complexity [6], [7], [8]. The proposed cognitive-spatial complexity measures take into account both the control structures as well as the spatial distances between program entities. Thus, the proposed measures are more suitable than both of the earlier approaches of cognitive complexity and spatial complexity measures. In order to compare the proposed measures with the existing cognitive and spatial measures, let us conduct a comparative study using a program first written in Java and then, C++ language.

1. Class Stack
2. {
3. int stack[];
4. int top;
5. Stack(int size) {
6. 
7. stack=new int [size];
8. top=-1;
9. }
10. void push(int item) {
11. 
12. if (top++ = = stack.length-1)
13. System.out.println("Stack full");
14. else
15. stack[top+1]=item;
16. }
17. int pop() {
18. 
19. if (top = = 0) return 0;
20. 
21. System.out.println("Stack Underflow");
22. return 0;
23. }
24. else
25. return stack[top--];
26. }
27. int isStackEmpty() {
28. { } return top = = -1;
29. } //isStackEmpty() method
30. int topOfStack() {
31. { 
32. }
33. return stack[top-1];
34. } //topOfStack() method
35. public static void main(String str[])
36. {
37. Stack s1=new Stack(5);
38. for(int i=0; i < 5; i++)
39. s1.push(i);
40. for(int i=0; i < 5; i++)
41. System.out.println(s1.pop());
42. } //main method
43. } //Class Stack

Figure 1 shows the Java implementation of the program and below Figure 2 shows its implementation in C++.
We compare the proposed measures with cognitive complexity measure (CFS) [5] and spatial complexity measures (CSC and OSC) [11] for programs shown in Fig. 1 and Fig. 2 and results of the comparison are shown in following Table 1.

<table>
<thead>
<tr>
<th>Table 1: Computation Results for the Measures</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>OO Cognitive-Spatial Complexity Measures</strong></td>
</tr>
<tr>
<td>CCSC</td>
</tr>
<tr>
<td>Java Program (Fig.1)</td>
</tr>
<tr>
<td>C++ Program (Fig.2)</td>
</tr>
</tbody>
</table>

Figure 3 depicts the graphical representations of the computed values of the measures for Java and C++ programs.

VI. CONCLUSIONS

In this paper, we have proposed new cognitive-spatial complexity measures for object-oriented software. The proposed measures take spatial as well as architectural complexity of the software into account for the estimation of the cognitive effort required for software comprehension process. The spatial complexity has been taken into consideration using the lexical distances (in LOC) between different program elements and architectural complexity of the software has been taken into account using the cognitive weights of various control structures present in the control flow of the program. Moreover, the proposed measures have been validated using standard axiomatic frameworks given by Weyuker [15] and Briand et al. [16] to prove their usefulness. Further, the proposed measures have been compared with existing cognitive and spatial complexity measures for object-oriented software. This comparative study has shown that the proposed measures are better indicators of the cognitive effort required for program comprehension than the corresponding existing cognitive and spatial complexity measures. In future work, we plan to conduct a more detailed empirical study over software of various sizes to judge their suitability for estimating the maintenance efforts of the software.

REFERENCES

Jitender Kumar Chhabra
Jitender Kumar Chhabra, PhD, is working as Assistant Professor in Department of Computer Engineering, National Institute of Technology (Deemed University), Kurukshetra-136119 (INDIA). He received his B.Tech. in Computer Engineering as 2nd rank holder and M.Tech in Computer Engineering as Gold Medalist, both from Regional Engineering College, (now N.I. T.) Kurukshetra. He completed his PhD degree on Software Metrics from GGS Indraprastha University, Delhi (INDIA). He is teaching in N.I.T. Kurukshetra since last 14 years. He has also worked in collaboration with companies like Hewellet-Packard & Tata Consultancy Services. He has published more than 50 research papers in various international and national journals and conferences including IEEE, Elsevier, ACM. He is reviewer of many reputed research journals like IEEE and Elsevier. He is adaptation author of Gottfried's Scham-Series book on Programming with C from Tata McGraw Hill. His areas of interest include software engineering, database system, data structure, programming techniques.

Varun Gupta
Varun Gupta is pursuing his Ph.D. degree in the area of Software Engineering from Department of Computer Engineering, National Institute of Technology (Deemed University), Kurukshetra-136119 (INDIA). He obtained his Bachelor of Technology degree in computer science & engineering from Guru Nanak Dev University, Amritsar in 1999 and Master of Engineering degree in software engineering from Thapar Institute of Engineering and Technology, Patiala (Deemed University) in 2003. He worked as a lecturer in Department of Computer Science & Engineering, RIMT Institute of Engineering and Technology for 4 years. Presently, he is working as Assistant Director in Directorate of Information Technology, PSEB, Patiala. His areas of interest include Software Engineering, Object Oriented Design & Development, and Data Mining.

International Journal of Computer and Information Engineering
Vol:3, No:3, 2009