Notice of Violation of IEEE Publication Principles

"Towards a Unified Framework for Cohesion Measurement in Aspect-Oriented Systems,"
by A. Kumar, R. Kumar, P.S. Grover
in the Proceedings of the 19th Australian Conference on Software Engineering. ASWEC 2008.
pp.57-65, March 2008

After careful and considered review of the content and authorship of this paper by a duly constituted expert committee, this paper has been found to be in violation of IEEE's Publication Principles.

This paper contains significant portions of original text from the paper cited below. The original text was copied without attribution (including appropriate references to the original author(s) and/or paper title) and without permission.

Due to the nature of this violation, reasonable effort should be made to remove all past references to this paper, and future references should be made to the following article:

"Towards a Unified Coupling Framework for Measuring Aspect-oriented Programs"
by T.T. Bartolomei, A. Garcia, C. Sant'Anna, C., E. Figueiredo
in the Proceedings of the 3rd International Workshop on Software Quality Assurance (Portland, Oregon, November 06 - 06, 2006). SOQUA '06. ACM

Towards a Unified Framework for Cohesion Measurement in Aspect-Oriented Systems

Avadhesh Kumar
Amity Institute of Information Technology
Amity University
Noida, India
avadheshkumar@aiit.amity.edu

Rajesh Kumar
School of Mathematics & Computer Applications
Thapar University
Patiala, Punjab, India.
rakumar@tiet.ac.in

P.S. Grover
Guru Tegh Bahadur Institute of Technology
GGS Indraprastha University
Delhi, India
groverps@hotmail.com

Abstract

Aspect-Oriented Programming (AOP) is an emerging technique that provides a means to cleanly encapsulate and implement aspects that crosscut other modules. However, despite an interesting body of work for measuring cohesion in Aspect-Oriented (AO) Systems, there is poor understanding of cohesion in the context of AOP. Most of the proposed cohesion assessment framework and metrics for AOP are for AspectJ programming language. In this paper we have defined a generic cohesion framework that takes into account two, the most well known families of available AOP languages, AspectJ and CaesarJ. This unified framework contributes in better understanding of cohesion in AOP, witch can contribute in (i) comparing measures and their potential use, (ii) integrating different existing measures which examine the same concept in different ways, and (iii) defining new cohesion metrics, which in turn permits the analysis and comparison of Java, AspectJ and CaesarJ implementations.

Keywords*: cohesion, aspect-oriented programming, software metric.*

1. Introduction

Software quality refers to the conformance of the product to explicitly stated functional and performance requirements, documented development standards, and implicit characteristics. Software quality is characterized by certain attributes, which are highlighted by some standards. An example of such standard is ISO/IEC 9126. Main characteristics of ISO/IEC 9126 are *functionality, maintainability, usability, efficiency, reliability,* and *portability* [1], [2]. Software development industry has been forced to place much more emphasis on software quality. In turn, researchers and practitioners have proposed a large number of new measures and assessment frameworks for quality design principles such as cohesion. High quality of any software means high cohesion.

Aspect-oriented programming (AOP) [3] languages aim to improve the ability of designers to modularize concerns that cannot be modularized using traditional module-oriented (MO) or object-oriented (OO) methods [4]. Such concerns are known as *crosscutting concerns.* Examples of *crosscutting concerns* include tracing, logging, caching, resource pooling and so on. The ability to modularize such concerns is expected to improve comprehensibility, parallel development, reuse and ease of change [5], [6], reducing development costs, increasing dependability and adaptability. Since AO is a new abstraction, the definition of cohesion is required to redefine in the context of AOP.

The most popular AOP model today is as implemented in AspectJ [7]. AspectJ extends Java with several complementary mechanisms, namely *join points* (JPs), *pointcut descriptors* (PCDs), *advice, introduction* and *aspects*. JPs represent well-defined points in a program's execution. Typical *join points* in AspectJ include method calls, access to class members, and the execution of exception handler blocks. A PCD is a language construct that picks out a set of *join points* based on defined criteria. The criteria can be explicit function names, or function names specified by wildcards. *Advice* is code that executes *before, after,* or *around* a join point. You define advice relative to a *pointcut*, saying something like "run this code before every method call I want to log”. *Introduction* allows aspects to modify the static structure of a program. Using introduction, aspects can add new methods and variables to a class, declare that a class implements an interface, or convert checked to unchecked exceptions. Advice, pointcuts and ordinary data members and methods are grouped into class-like modules called *aspects*. Aspects are intended to support the modular representation of crosscutting concerns [8], although they admit other uses. Some existing AOP languages and frameworks provide a very similar composition model to the AspectJ one, such as Springs AOP framework [9] and JBoss AOP [10]. However, despite an interesting body of work for measuring cohesion in AO Systems, there is poor understanding of cohesion in the contest of AOP. Some researchers and practitioners [11], [12], [13], [14], [15] have proposed cohesion assessment framework and metrics for AOP. But most of the framework and metrics are for AspectJ programming language. They have defined cohesion in their framework and metrics in the contest of AspectJ programming language.

On the other extreme, CaesarJ [16] represents a different family of AOP languages since it does not have aspects as a separate language abstraction, but supports additional concepts such as *virtual classes*, *mixin composition*, *aspectual polymorphism*, and *bindings*. Despite the growing body of work dedicated to measure cohesion in AO systems, there is no unified measurement framework for characterizing what constitutes cohesion in AO systems.

In this paper, we have proposed the description of a unified cohesion measurement framework that builds on top of existing cohesion measurement frameworks and measures for AO System. This proposed cohesion measurement framework for AO systems refines Briand's [17] framework and has specifically targeted at the composition models supported by Java, AspectJ and CaesarJ. This will be helpful in (i) comparing measures and their potential use, (ii) integrating different existing measures which examine the same concept in different ways, and (iii) defining new cohesion metrics, which in turn permits the analysis and comparison of Java, AspectJ and CaesarJ implementations.

The paper is structured accordingly. Section 2 defines new terminology used for AO systems. Section 3 presents a critical review of existing cohesion measurement frameworks and measures. After defining new terminology, we have proposed unified framework for cohesion measurement in section 4. In section 5, applications of proposed framework are discussed. Conclusions and future work are presented in section 6.

2. Terminology and Formalism

To define cohesion measures it is necessary to establish a proper standardized, unambiguous and operational terminology and formalism. Briand et. al. [17] introduced a terminology for object oriented (OO) systems based on set and graph theory. In this section we introduce the terminology and formalism foundation for structural elements in an AO system. The presented terminology is for Java, AspectJ-like, and CaesarJ, programming languages.

2.1 Structure

Aspect-Oriented (AO) systems structure is a set of fundamental elements, which defines its key abstractions for module specifications. Figure 1 [18] presents a detailed view of the structural abstractions defined for a system. Each type of abstraction is alternatively called an element.

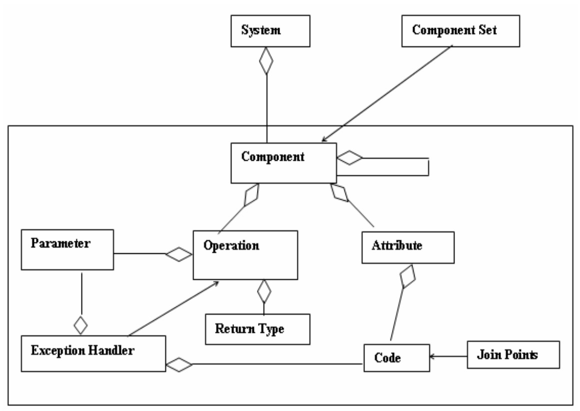

Figure 1

An AO system S consists of a set of components, denoted by C(S). A component c consists of a set of attributes, Att(c), a set of operations, Op(c) and a set of nested components, Nested(c). In other words, a component is composed of subcomponents in addition to its internal attributes and operations. The set of members of a component c is defined by M(c) = Att(c) ∪ Op(c) ∪ Nested(c). The reader should notice that for generality purposes, a component is an unified Figure 1: Structure of an Aspect-Oriented System abstraction to both aspectual and non-aspectual modules. For example: (i) a component represents either a class or an interface in Java programs, (ii) a component is either a class, an interface, or an aspect in AspectJ programs, and (iii) a component is either a cclass as defined in CaesarJ, a conventional class, or an interface in CaesarJ programs. An operation o consists of a set of code, Code(o), a set of parameters, Par(o) and a return type, Rt(o). An attribute a consists of a set of code, Code(a). Each piece of code cod provides a set of join points, JP(cod).

2.2 System

Definition 1: An AO system S consists of a set of components denoted by C(S).

2.3 Components

Definition 2: A component c consists of attributes, *Att*(c), A set of operations, *Op*(c), and a set of nested components, *Nested*(c). The set of members of a component c is defined as:

$$M(c) = Att(c) \cup Op(c) \cup Nested(c).$$

This abstract structure can be instantiated for different languages. In this paper, we have considered Java (J), AspectJ (AJ) and CaesarJ (CJ) programming languages. Abstractions that map to all languages are marked with an *All.* Aspects and Inter-type declarations are only valid for AspectJ-like languages, CClasses are defined only for CaesarJ language. So component is Class (*All),* Interface *(All),* Aspect *(AJ)* and CClass (*CJ).* There can be inheritance relationship between components such that for each component $c \in C$ let

- *Parents(c)* $\subset$ C be the set of parent components of component c.
- *Children(c)* $\subset$ C be the set of children components of component c.
- *Ancestors(c)* $\subset$ C be the set of ancestor components of component c.
- *Descendents(c)* $\subset$ C be the set of descendent components of component c.

Because of the inheritance relationships between components, the following member sets are defined for a component c:

- $M_{NEW}(c)$ - Members newly implemented in the component (not overridden).
- $M_{VIR}(c)$ - Members declared as virtual in the component.
- $M_{OVR}(c)$ - Members inherited and overridden.
- $M_{INH}(c)$ - Members inherited and not-overridden.

2.4 Operations

Definition 3: An operation o consists of a set of code, *Code* (o), a set of parameters, *Par* (o), and a return type, *Rt* (o). The set of members of an operation o is defined as:

$$M(o) = Code\ (o) \cup Par\ (o) \cup Rt\ (o).$$

For this abstract structure, an operation is Method (*All*), Constructor (*All*), Static Initializer (*All*), Intertype Method (*AJ*), Intertype Constructor (*AJ*), Intertype Attribute (*AJ*), Pointcut (*AJ*, *CJ*), Advice (*AJ*, *CJ*), Declare Statements (*AJ*, *CJ*) and Wrapper Constructor(*CJ*).

Definition 4: Declared and Implemented operations
For each component $c \in C$, let

O(c) be the set of operations of component c.

- $O_D(c) \subseteq O(c)$ be the set of operations *declared* in c, i.e., operations that c inherits but does not override or virtual operation of c.
- $O_I(c) \subseteq O(c)$ be the set of operations *implemented* in c, i.e., operations that c inherits but overrides or non-virtual operation of c.

Where $O(c) = O_D(c) \cup O_I(c)$ and $OD(c) \cap O_I(c) = \phi$.

Definition 5: Inherited, overriding and new Operations
For each component $c \in C$ let

- $O_{NEW}(c) \subseteq O(c)$ be the set of newly implemented operations in the component c (not overridden).
- $O_{VIR}(c) \subseteq O(c)$ be the set of operations declared as virtual in the component c.
- $O_{OVR}(c) \subseteq O(c)$ be the set of operations inherited and overridden in the component c.
- $O_{INH}(c) \subseteq O(c)$ be the set of operations inherited and not-overridden in the component c.

Definition 6: Public and non-public operations
For each component $c \in C$ let

- $O_{PUB}(c) \subseteq O(c)$ be the set of public operations of c and
- $O_{NPUB}(c) \subseteq O(c)$ be the set of non-public operations of c.

Where $O(c) = O_{PUB}(c) \cup O_{NPUB}(c)$ and $O_{PUB}(c) \cap O_{NPUB}(c) = \phi$.

A public operation can be accessed by other operation of the system, while access of non-public operation varies from programming language to programming language. We are taking no assumption for the access of non-public method in our paper.

Definition 7: Set of all operations in the system

O(C) is the set of all operations in the system and is represented as $O(C) = \cup_{c \in C} O(c)$.

2.5 Operation Invocations

In AO system, operation invocation can be defined in two categories: (i).*Implicit Invocations are* caused by a join point being reached or by an exception handler catching an exception, and (ii) *Explicit Invocations* are caused by code calling the operation. Method invocation can be either static or dynamic; it is necessary to distinguish between these. Following are the definitions for the types of invocations.

Definition 8: $IOI_{ST}(o)$ - implicit operation invocations that can be fully statically determined.
Definition 7: $IOI_{DYN}(o)$ - implicit operation invocations that can be only dynamically determined.
Definition 7: $EOI_{ST}(o, o')$- explicit operation invocations that can be fully statically determined.
Definition 7: $EOI_{ID}(o)$- explicit operation invocations, accounting for implementing descendants of the operations component.

2.6 Indirect Operation Invocations

For an operation $o \in O(C)$, $EOI_{ST}(o, o')$ and $EOI_{ID}(o)$ are sets of operations explicitly directly invoked by o. There is need to account those operations also which can be indirectly invoked by o. Operation o indirectly invoked o', if there are operations $o_1, o_2 \ldots o_n$ such that o directly invokes o_1, o_1 directly invokes o_2, etc., and o_n directly invokes o'. This could be defined as:

Definition 8: Indirectly invoked operations
$\forall o \in O(C)$, let

$EOI_{ST}(o, o') = \{o' \mid o' \in O(C) \wedge \exists n \geq 1 \exists o_1, o_2 \ldots o_n \in O(C): o_1 = o \wedge o_n = o' \wedge \forall i, 1 < i \leq n: o_i \in EOI_{ST}(o_{i-1}, o')$

$EOI_{ID}(o) = \{o' \mid o' \in O(C) \wedge \exists n \geq 1 \exists o_1, o_2 \ldots o_n \in O(C): o_1 = o \wedge o_n = o' \wedge \forall i, 1 < i \leq n: o_i \in EOI_{ID}(o_{i-1})$

2.7 Attributes

Component have either inherited or newly defined attributes. Attributes could be defined as:

Definition 9: Declared and implemented attributes
For each component $c \in C$ let A(c) be the set of attributes of component c. $A(c) = A_D(c) \cup A_I(c)$ where

- $A_D(c)$ is a set of attributes declared in component c (i.e., inherited attributes)
- $A_I(c)$ is a set of attributes implemented in component c (i.e., new added attributes)

Definition 10: A(C) the set of all attributes
A(C) is the set of all attributes in the system and is represented as $A(C) = \cup_{c \in C} A(c)$.
In language specification attribute is either a Field (*All*) or Intertype Field (*A*).

2.8 Attribute References

Attribute references may also be influenced by polymorphism. Hence, a complementary list of sets of attribute references AR(c) is defined for a component c.

Definition 11:

- $AR_{ST}(c)$ – is the set of attribute references, accounting only for statically determined attributes.
- $AR_{ID}(c)$ – is the set of attribute references, accounting for implementing descendants of the attribute's component.
- AR_{FI} (c) – is the set of attribute references, accounting only for the first statically determined implementation, looking for the attribute's component and its ancestors.
- AR_{ST} (c, c') we define sets with a meta variable representing the target component c'.

2.9 Types

Attribute and operation have some type and return type respectively.

Definition 12: Built-In types, User-Defined types and Component types
Let

- BT(S) be the set of built-In types of the system S.
- UDT(S) be the set of user defined types of the system S.
- CT(S) be the set of component types of the system S.

T(S) be the set all available types then $T(S) = BT(S) \cup UDT(S) \cup CT(S)$.

3. Survey of Cohesion Measurement Frameworks and Measures

There are many assessment framework and measurement approaches for cohesion measurement in OO Systems. It is because of high maturity level of OO paradigm. AOP is a new and emerging paradigm. In literature, there exist few assessment framework/measurement approach/metrics [11], [12], [13], [14], [15] to measure cohesion in AO systems.

And most of the measures are for AspectJ programming language. There is no generic cohesion measurement framework.

3.1 Approach by Zhao and Xu [12]

First approach towards measuring aspect cohesion was by Zhao and Xu. Their approach is based on a dependency model for aspect-oriented software that consists of a group of dependency graphs. Zhao and Xu defined cohesion as the degree of relatedness between attributes and modules (method/advice). Zhao and Xu present, in fact, two ways for measuring aspect cohesion based on inter-attributes (γa), inter-modules (γm) and module-attribute (γma) dependencies. The first way suggests that each measurement (γa, γm, γma) works as a field. Therefore, aspect cohesion for a given aspect A is defined as a 3-tuples Γ(A) = (γa, γm, γma). They also suggested another way for cohesion measurement where each facet could be integrated as a whole with β parameters. Aspect cohesion is then expressed as:

$$\Gamma(A) = \begin{cases} 0, & n = 0 \\ \beta^* \gamma m, & k = 0 \text{ and } n \neq 0 \\ \beta 1^* \gamma a + \beta 2^* \gamma m + \beta 3 ^* \gamma ma, & \text{others} \end{cases}$$

Where k is the number of attributes and n is the number of modules in aspect A. $\beta \in (0,1)$, β1, β2, β3 >0, and β1+ β2+ β3=1.

Users determine the selection of weight β1, β2, and β3, which is arbitrary. In addition, some relationships definition (inter-attributes cohesion in particular) and their consideration in aspect cohesion measurement are difficult to capture. In general, this approach suggests a complex way to measure aspect cohesion that may be problematic to use in a real development context and particularly in the case of real scalable aspect-oriented software. Generating such dependency graphs is a time consuming process while parameter's weighting could be misleading. Secondary, using this approach, cohesion measurement is only for the family of AspectJ programming languages.

3.2 Framework by Sant'Anna, Garcia, Chavez, Lucena & Staa [13]

Sant' Anna et al. proposed a new metric LCOO (Lack of Cohesion in Operations) measures the amount of method/advice pairs that do not access to the same instance variables. It is an extension of the well-known LCOM (Lack of Cohesion in Methods) metric developed by Chidamber and Kemerer [19]. This metric measures the lack of cohesion of a component (class and aspect). According to their approach, consider a component C1 with operations (methods and advice) Oi, …, On. Let {Ij}= set of instance variables used by operation O_j. There are n such sets $\{I_1\}, \{I_2\}, \ldots, \{I_n\}$. Let

$|P| = \{(I_i, I_j) \mid I_i \cap I_j = \acute{\emptyset}\}$, and
$|Q| = \{(I_i, I_j) \mid I_i \cap I_j \neq \acute{\emptyset}\}$

If all n sets $\{I_1\}, \ldots, \{I_n\}$ are $\acute{\emptyset}$ then let P=Q.

$$\text{LCOO} = \begin{cases} |P| - |Q|, & \text{if } |P| > |Q| \\ 0, & \text{otherwise.} \end{cases}$$

A high LCOO value, according to Sant' Anna et al., indicates disparateness in the functionality provided by the aspect. The definition of LCOO metric is almost operational. It is not stated whether inherited operations and attributes are included or not, and we have to assume that sets $\{I_j\}$ only include attributes of component C1. It suffers from several problems as stated, among others, by B. Henderson-Sellers in [20]. Again, definition of this metric is based on AspectJ-like programming languages.

3.3 Approach by Gelinas, Badri, and Badri [14]

Gelinas et al. proposed a new metric ACoh to measure aspect cohesion. This cohesion measurement is based on dependency analysis. They defined this metric using two aspect cohesion criteria, *Modules-Data Connection Criterion* and *Modules-Modules Connection Criterion.*

3.3.1 Modules-Data Connection Criterion

Let UA_{Mi} be the set of attributes used directly or indirectly by the module Mi. An attribute is used directly by a module M_i if the attribute appears in its body. An attribute is indirectly used by a module M_i if it is used directly by another module invoked directly or indirectly by M_i. There are m sets UA_{M1}, UA_{M2}, … UA_{Mm}. Two modules M_i and M_j of an aspect are related by the *UA_M relationship* if $UA_{Mi} \cap UA_{Mj} \neq \emptyset$. It means that there is at least one attribute shared (directly or indirectly) by the two modules.

3.3.2 Modules-Modules Connection Criterion

Let UM_{Mi} be the set of modules used directly or indirectly by the module M_i. A module M_j is used directly by a module M_i if M_j appears in the body of M_i. A module M_j is indirectly used by a module M_i if it is used directly by another module or indirectly by M_i.

There are m sets UM_{M1}, UM_{M2}, … UM_{Mm}. Two modules M_i and M_j of an aspect are related by the *UM_M relation* if $UM_{Mi} \cap UM_{Mj} \neq \emptyset$. It means that there is at least one module jointly used (directly or indirectly) by the two modules. They also consider that M_i and M_j are directly related if $M_j \in UM_{Mi}$ or $M_i \in UM_{Mj}$.

3.3.3 Aspect Cohesion Measurement

Two modules M_i and M_j can be connected in many ways: by sharing attributes or by sharing modules or both. Let $NM(Aspect_i)$ be the total number of modules pairs in an aspect. NM is the maximum number of connections between aspect modules. Thus in an aspect having N modules, $NM(Aspect_i)=N*(N-1)/2$, $N>1$. $NC(Aspect_i)$ be the number of connection between modules in undirected graph G_D. They defined a new metric for aspect cohesion measures:

$$ACoh(Aspect_i) = NC(Aspect_i)/NM(Aspect_i) \in [0, 1].$$

This cohesion measure metric is based on AspectJ-like programming languages. It is not clear, whether they have accounted inherited member or not. If accounted, what will be the affects on cohesion values?

3.4 A Framework by Sant'Anna, Figueiredo, Garcia and Lucena [15]

In this framework Sant'Anna et al. defined a new metrics suite for concern-driven architecture. The proposed metric suite is to measure cohesion, coupling and complexity of AO and non-AO systems. The proposed metric LCC (Lack of Concern-based Cohesion) for measuring cohesion in concern-driven architecture counts the number of concerns addressed by the assessed component. This is very useful framework for evaluating internal software quality attributes such as cohesion, coupling, complexity and modularity at architecture level. It is not specified, how to quantify aspect cohesion for empirical evaluation. It is also not clear whether inheritance has been accounted for or not.

4. A Unified framework for Cohesion Measurement

To define a unified cohesion measurement framework, we have analyzed frameworks for OO systems [17], AO systems [12], [13], and specific for AspectJ [14]. Also we have identified the influences of CaesarJ and it's specific features. From these analyses a list of criterion that must be considered when proposing a new cohesion measurement framework emerged. These criterions are supposed to answer the questions. The cohesion criterions that we have considered in this proposed framework are originally defined in Briand's [14] frameworks.

Our framework consists of five criterions; each criterion determines one basic feature of the resulting measure. First we describe each criterion, what decisions have to be made, what are the available options, how the criterion reflected by cohesion measures, and how the framework can be used to derive cohesion measures.

4.1 Type of connections

What constitutes cohesion?

This criterion defines mechanism that causes cohesion in component. A connection within a component is a link between elements of the component (operation, attribute, inter-type attribute, or data declarations). In this framework we take into account connections related to OO cohesion [17], AspectJ specific connections [14], and generic AO connections [12]. We also study possible connections for cohesion caused by CaesarJ type constructs.

Definition 13: Type of connection of a component c
Let
An operation $o, o' \in O(c)$ and an attribute $a \in A(c)$

- CON# 1: *o* references *a*..
- CON# 2: *o* invokes *o'*.
- CON# 3: *o* and *o'* directly reference *a of* c in common ("similar operations").
- CON# 4: *o* and *o'* directly or indirectly reference *a of* c in common ("connected operations").
- CON# 5: data-data interaction (data declaration in c)
- CON# 6: data-operation interaction

Where CON#, represents connection number, which we have used as reference in other sections.

4.2 Domain of measure

Which element to account for, which to restrict?

By domain of measure means finding out which elements (attribute, operation, etc.) will participate in determining cohesion of the component. Most of the reviewed measures are defined at the component level. However, finer and coarser domains are also conceivable. We have also considered new elements (cclass, wrapper constructor etc.) of CaesarJ family. We can count the number of other component elements to which it is connected for an individual attribute or operation. This is analyzing how closely related the attribute or operation is to other elements of its component or in other words it is the degree to which the attribute or operation contributes to the cohesion of its component. From this measure, we can analyze which element of the component should remain in the same component and which should be moved to new or another component.

We can also quantify the cohesion of component set or system based on the cohesion of each participating component. It is also required to select the types of elements that should be accounted for. For example Gelinas et. al. [14] counts only attributes (data members) and module (methods and advice). They measure cohesion on the basis of two connection criterion, modules-data connection criterion and modules-modules connection criterion. Zhao & Xu [12] measured aspect cohesion in which they accounted attribute, intertype attribute, intertype methods, and advice as members responsible for cohesion of the component. In this proposed framework we have accounted attributes and operations defined in section 2.4, section 2.7, and section 2.8, which includes elements of class (Java), aspect (AspectJ), and cclass (CaesarJ). CaesarJ is new abstraction family of which has been included in this framework. Table -I shows domain of measures with terminology representation.

Table-I: Domain of measures with terminology representation.

Domain of Measures	Terminology Representation
Components	Class *(All),* Interface *(All),* Aspect *(AJ)* and CClass (*CJ)*
Operations	Method (*All*), Constructor (*All*), Static Initializer (*All*), Intertype Method (*AJ*), Intertype Constructor (*AJ*), Intertype Attribute (*AJ*), Pointcut (*AJ*, *CJ*), Advice (*AJ*, *CJ*), Declare Statements (*AJ*, *CJ*) and Wrapper Constructor(*CJ*).
Attributes	A_D(c) and A_I(c)

4.3 Direct or indirect connections

Count direct connections only or also indirect connections?

In this section our focus is whether to count direct connections only or also indirect connections. Direct connections definitely makes a component cohesive (All connection types CON# 1 to CON# 6). But for indirect connection cohesion measures depends on the type of connections. For example consider an operation o_1, which is "similar" to another operation o_2 (CON#3), operation o_2 is "similar" to operation o_3 (CON#3). Operations o_1 and o_2 are directly connected. Similarly operations o_2 and o_3 are also directly connected. Now we can say operations o_1 and o_3 are indirectly connected, which is a transitive relationship connection.

4.4 Inheritance

Whether to account or not the members of Parents(c), and Ancestors(c) of component c?

There are two different aspects to be considered with respect to inheritance.

- How do we assign attributes and operations to components?
- Whether to consider static or polymorphic invocation of operation?

These could be described as:

How do we assign attributes and operations to components?

For the analysis of cohesion measures, there are two elements of a component c, attributes A(c) and operations O(c). For the measures of cohesion, either we can account non inherited elements only or also account inherited elements.

1. *Exclude inherited attributes and operations from the analysis.*

If we exclude inherited attributes and operations, cohesion analysis will be to the same component only and will represent a single semantic concept.

2. *Include inherited attributes and operations in the analysis.*

If we include inherited attributes and operations, cohesion measures accounts attributes inherited, attributes newly added and operations O_{NEW}(c), O_{VIR}(c), O_{OVR}(c), and O_{INH}(c) in the component c. We will analyze whether component c as a whole still represents a single semantic concept.

Whether to consider static or polymorphic invocation of operation?

Polymorphism will be relevant only if the type of connection involves operation invocation. If c is a component, and *Ancestors(c)* is/are the ancestor(s) of c, then we have following two options:

1. *Do not account for polymorphism.*

Let o be an operation of component c, which is having connection with operations o'. o' can possibly be invoked statically in the implementation of o.

2. *Account for polymorphism.*

Let o be an operation of the component c, which is having connections with operations o'. o' can possibly be in the implementation of o through polymorphism and dynamic binding.

4.5 Access operations and constructors

How to account for access operations and constructors for measuring cohesion?

Access operations and constructors may artificially increase or decrease cohesion measure values. In the definition of cohesion measures and measurement framework, we have to decide how to account access operations and constructors.

Access operations
Following options can be used to account for access operations

1. *Treat access operations as regular operations*

Do nothing; consider access operations as regular operations, which may cause artificially increase or decrease in cohesion values.

2. *Consider the invocation of an access operation as a reference to the attribute*

If an operation is not referring an attribute directly, an access operation could be used to refer the attribute indirectly. These access methods are not accounted for by CON#1 & CON#3, which causes artificially decreased number of references to attributes. If we account the invocation of access operation as reference to the attributes, it will be solution to this problem. Implementation of this problem is difficult, because to recognize access operations automatically is not always possible.

3. *Exclude the access operations from the analysis*

If pair of operations are using common attributes (CON#3 & CON#4), then access operations may cause problems for cohesion measures. Access operations usually access only one attribute, many pairs of operations that don't refer a common attribute can be found using access operation. This problem causes artificially decrease in cohesion. Solution to this problem is to exclude access operations from the analysis.

Constructors
Although we have considered constructors (*All*) and Wrapper Constructors (*CJ*) as part of operation in section 2.4, even than it is important to discuss effects of constructor and wrapper constructors separately. Either we can include constructors in cohesion analysis or we can exclude constructors from cohesion analysis.

1. *Include constructors in the analysis*

Do nothing, count constructors and wrapper constructors as operation like method and advice.

2. *Exclude constructors from the analysis*

Constructors and wrapper constructors cause problems for measures that count pairs of operations which use common attributes (CON#3 and CON#4). Most of the time constructors typically reference all the attributes. It causes artificially increased in cohesion of the component, because it generates many pairs of operations that use a common attribute. The solution to this problem is to exclude constructors from the analysis.

5. Framework Application

In this section we have proposed a generic cohesion metric as an application of our proposed generic/unified cohesion measurement framework. This metric is an extension of existing metric defined by Gelinas et.al [14]. They defined it for AspectJ-like programming language, which is ACoh (Aspect_i) = number of actual connections of interest/ maximum number of possible connection of interest. First, we go through the whole process of analyzing and selecting the criteria. Second, we derive a formal definition.

5.1 Criteria Selection

We have accounted Java, Aspectj-like and CaesarJ-like languages in our framework. For each criteria of the framework, choose one or more of the available options depending on the decisions on the objective of the measurement.

Definition of the new metric is for AOP. Hence, the types of connections are targets of explicit invocations and targets of join points implicit invocations. Domain of measures will be components, operations, and attributes. Account both direct and indirect connections because both involves in the cohesion of the component. Include inherited attributes and operations in the analysis, and also account for polymorphism. Treat access operations depending on conditions specified in section 4.5. Exclude constructors from the analysis.

5.2 Formal Definition

Using the AO terminology given in section 2 and selection criteria we derive a formal definition of unified aspect cohesion metric UACoh (c_i) for a component i (c_i) of the system S.

$$\text{UACoh}(c_i) = \text{ANC}(c_i) / \text{MNC}(c_i) \in [0, 1].$$

Where ANC (c_i) is the actual number of connections between members (CON#1 to CON#6) of c_i and MNC (c_i) is the maximum number of possible connections of interest between the members (CON#1 to CON#6) of interest of c_i.

6. Conclusions and Future Work

In this paper we have proposed a unified framework for cohesion measures in AO systems. We have accounted two families of AOP languages: AspectJ-like languages and another family of languages which also support feature oriented decomposition mechanism such as CaesarJ. In this framework we have defined new AO terminology. Using this terminology and cohesion selection criteria defined in the framework, we also defined new unified aspect cohesion metric (UACoh(c_i)) of a component i, for generic AO systems. This proposed metric is an extension of an existing metric, which is defined for AspectJ programming language. Proposed cohesion metric is generic when selection criteria for selecting members will be applied properly.

Our framework has already been used to describe existing cohesion metrics and measures. However, the framework lacks empirical validation. In future work, we have planned to validate the framework using different AOP languages. First step to validate framework will be the empirical validation of proposed AO cohesion metric based on the selection criteria. Cohesion has a large impact on external software characteristics such as maintainability, reusability and modularity, so by using this cohesion metric, we can evaluate external software characteristics of generic AO systems. Second step will be to evolve up to what extend our framework is generic enough to define in symmetric AO languages such as Hyper/J. Results seems to show that to make the framework generic, major extensions and changes will not be required. We are working in this direction.

7. References

[1]. Jorgen Boegh, Stefano Depanfilis, Barbara Kitchenham, Alberto Pasquini, "A Method for Software Quality Planning, Control, and Evaluation", Software IEEE Journal, vol-16, issue-2, pp.69-77, March-1999.

[2]. Ho-Won Jung; Seung-Gweon Kim; Chang-Shin Chung," *Measuring software product quality: a survey of ISO/IEC 9126*", Software IEEE,Volume 21, Issue 5, pp.88-92 , Sep-Oct-2004.

[3]. K. Lieberher, D. Orleans, and J. Ovlinger, "*Aspect-Oriented Programming with Adaptive Methods*," *Communications of the AC*M, Vol.44, No.10, pp.39-41, October 2001.

[4]. Avadhesh Kumar, Rajesh Kumar, P.S. Grover, "*A Comparative Study of Aspect-Oriented Methodology with Module-Oriented and Object-Oriented Methodologies*", Icfai Journal of Information Technology, Vol. 2, No. 4, pp.7-15, December 2006.

[5]. Avadhesh Kumar, Rajesh Kumar, P.S. Grover, "*A Change Impact Assessment in Aspect-Oriented Software Systems*", International Software Engineering Conference Russia 2006 (SECR-2006), pp.83-87, Dec 2006.

[6]. Avadhesh Kumar, Rajesh Kumar, P.S. Grover, "*An Evaluation of Maintainability of Aspect-Oriented Systems: a Practical Approach*", International Journal of Computer Science and Security, Vol -1, Issue-2, pp. 1-9, Aug 2007.

[7]. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. "*An Overview of AspectJ*". In Proceedings of the 15th European Conference on Object-Oriented Programming, pp. 327–355, Springer, 2001.

[8]. V. C. Garcia, E. K. Piveta, D. Lucrédio, A. Álvaro, E. S. Almeida, L.C. Zancanella, & A.F. Prado, "*Manipulating crosscutting concerns*" , Proc. 4th Latin American Conf. on Patterns Languages of Programming (SugarLoafPLoP), Porto das Dunas, CE, Brazil, 2004.

[9]. R. Johnson. Introducing the Spring framework., 2003. http://www.theserverside.com/tt/articles/article.tss?l=SpringFramework.

[10]. J. Inc. JBoss AOP Beta3, 2004. http://www.jboss.org.

[11]. M. Ceccato and P. Tonella. "*Measuring the Effects of Software Aspectization*", In Proceedings of the 1st Workshop on Aspect Reverse Engineering, 2004.

[12]. J. Zhao and B. Xu, "*Measuring Aspect Cohesion*", Proc. International Conference on Fundamental Approaches to Software Engineering (FASE'2004), LNCS 2984, pp.54-68, Springer-Verlag, Barcelona, Spain, March 29-31, 2004.

[13]. C. Sant'Anna, Alessandro Garcia, Christina Chavez, Carlos Lucena & Arndtvon Staa, "*On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework", XXIII Brazilian Symposium on Software Engineering*, Manaus, Brazil, October 2003.

[14]. J.F. Gelinas, M. Badri, L. Badri: "*A Cohesion Measure for Aspects*", in *Journal of Object Technology*, vol. 5, no. 7, pp. 97 – 114, September - October 2006.

[15]. Cláudio Sant'Anna, Eduardo Figueiredo, Alessandro Garcia, Carlos J. P. Lucena, "*On the Modularity Assessment of Software Architectures: Do my architectural concerns count?",* 6th International Workshop on Aspect-Oriented Software Development (AOSD'2007), Vancouver, British Columbia, March 12-13, 2007.

[16]. I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. "*Overview of CaesarJ*", Transactions on AOSD I, LNCS, 3880: pp.135 – 173, 2006.

[17]. Lionel C. Briand, John W. Daly, Jurgen Wust, "*A Unified Framework for Cohesion Measurement in Object-Oriented Systems*", Empirical Software Engineering : an International Journal, vol 3, no 1, pp. 65-117, 1998.

[18]. Thiago T. Bartolomei, Alessandro Garcia, Cláudio Sant'Anna, Eduardo Figueiredo, " Towards a Unified Coupling Framework for measuring Aspect-Oriented Programs", 3rd International Workshop on Software Quality Assurance (SOQUA 2006) Portland, Oregon, USA, November 6, 2006.

[19]. S. R. Chidamber and C. F. Kemerer., *"A Metrics Suite for Object-Oriented Design*", IEEE Transactions on Software Engineering, 20(6):pp.476–493, 1994.

[20]. B. Henderson-Sellers, "*Software Metrics*", *Prentice Hall,* Hemel Hempstaed, U.K., 1996.