Measuring Coupling in Aspect-Oriented Systems

Jianjun Zhao
Department of Computer Science and Engineering
Fukuoka Institute of Technology
3-30-1 Wajiro-Higashi, Higashi-ku, Fukuoka 811-0295, Japan
zhao@cs.fit.ac.jp

Abstract

Coupling is an internal software attribute that can be used to indicate the degree of interdependence among the components of a software system. Coupling is thought to be a desirable goal in software construction, leading to better values for external attributes such as maintainability, reusability, and reliability. Aspect-oriented software development (AOSD) is a new technique to support separation of concerns in software development. In aspect-oriented (AO) systems, the basic components are aspects or classes, which consist of attributes (aspect or class instance variables) and those modules such as advice, intertype declarations, pointcuts, and methods. Thus, in AO systems, the coupling is mainly about the degree of interdependence among aspects and/or classes. To test this hypothesis, good coupling measures for AO systems are needed. In this paper, we propose a coupling measure suite for assessing the coupling in aspect-oriented systems. We first present a coupling framework for AO systems which specially designed to count the dependencies between aspects and classes in the systems. Based on this framework, we formally define various coupling measures in terms of different types of dependencies between aspects and classes. We also discuss the mathematical properties of these measures.

1 Introduction

Aspect-oriented software development (AOSD) is a new technique to support separation of concerns in software development [3, 12, 14, 17]. The techniques of AOSD make it possible to modularize crosscutting aspects of a system. Like objects in object-oriented software development, aspects in AOSD may arise at any stage of the software life cycle, including requirements specification, design, implementation, etc. Some examples of crosscutting aspects are exception handling, synchronization, and resource sharing.

The current research so far in AOSD is focused on problem analysis, software design, and implementation techniques. However, efficient evaluations of this new design technique in a rigorous and quantitative fashion is still ignored during the current stage of the technical development. For example, it has been frequently claimed that applying an AOSD method will eventually lead to quality software, but unfortunately, there is little data to support such claim. Aspect-oriented software is supposed to be easy to maintain, reuse, and evolution, yet few quantitative studies on maintenance, reuse, and evolution have been conducted, and measures to quantify the amount of maintenance, reuse, and evolution in aspect-oriented software are lacking. In order to verify claims concerning the maintainability, reusability, and reliability of software developed using aspect-oriented techniques, software measurement tools are required.

As with object-oriented systems [16], we wish to be able to relate some internal attributes of an AO system such as aspect-oriented structural quality to some external attributes of the system such as maintainability, reusability, and reliability. We therefore need appropriate measures of aspect-oriented structure to begin to build such kinds of relations. Coupling is an internal software attribute that can be used to indicate the degree of interdependence among the components of a software system. It has been recognized that good software design should obey the principle of low coupling. A system that has strong coupling may make it more complex because it is difficult to understand, change, and correct highly interrelated components in the system. Recently, many coupling measures and several guidelines to measure coupling of a system have been developed for object-oriented systems [7, 10, 5, 4].

In an aspect-oriented (AO) system , the basic units are aspects and classes. An aspect with its encapsulation of state (attributes) with associated modules such as advice, inter-type declarations, pointcuts, and methods (operations) is a significantly different abstraction in comparison to the class within object-oriented systems. An aspect may also interact with one or more classes in many ways through advice, inter-type declaration, pointcut, and method call. Thus, in

AO systems, the coupling is mainly about the degree of interdependence among aspects and classes. To test this hypothesis, appropriate coupling measures for AO systems are needed. Moreover, in order to measure the coupling of an aspect-oriented system, we should consider different types of interactions between aspects and classes in the system.

However, although coupling has been widely studied for object-oriented systems [5, 4, 7, 10], it has not been studied for AO systems yet. Moreover, existing approaches to measuring the coupling of object-oriented systems can not be directly applied to aspect-oriented systems since an AO system contains some new types of interactions between aspects and/or classes that may have great impact on the coupling of the system. It is therefore of interest to develop new coupling framework and new coupling measures based on this framework for AO systems.

In this paper, we propose a measure suite for assessing the coupling in AO systems. We first present a coupling framework for AO systems which specially designed to count the dependencies between aspects and classes in the system. Based on this framework, we formally define various coupling measures in terms of different types of dependencies between aspects and classes. We also discuss the mathematical properties of these measures.

Because aspect-oriented paradigm significantly different from object-oriented paradigms, we really need to develop a notion of coupling for AO systems, which is an indicator of the degree to which the components in the system interact each other. We hope that by examining the ideas of the coupling in aspect-oriented systems from several different viewpoints and through independently developed measures, we can have a better understanding of what the coupling is meant in AO systems and the role that coupling plays in the development of quality aspect-oriented software. As the first step to study the coupling in aspect-oriented systems, in this paper we would like to provide a sound and formal basis for coupling measurement in AO systems before applying it to real aspect-oriented system design.

The rest of the paper is organized as follows. Section 2 briefly introduces AspectJ, a general aspect-oriented programming language based on Java. Section 3 defines a terminogy for an AO system. Section 4 presents a coupling framework for defining coupling measures for AO systems. Section 5 defines a coupling measure suite based on the framework presented in Section 4. Section 6 studies the mathematical properties of the proposed coupling measures. Section 7 discusses some related work. Concluding remarks are given in Section 8.

2 Aspect-Oriented Programming and AspectJ

In this paper we use AspectJ [2] as our target language to show the basic ideas of coupling measurement in AO systems. We believe that our ideas are independent of AspectJ and are generally applicable to the class of AOP languages.

AspectJ [13] is a seamless aspect-oriented extension to Java by adding some new concepts and associated constructs. These concepts and associated constructs are called join points, pointcut, advice, inter-type declaration, and aspect.

The *aspect* is the modular unit of crosscutting implementation. Each aspect encapsulates functionality that crosscuts other classes in a program. An aspect can be instantiated, can contain states and methods, and also may be specialized with subaspects. An aspect is combined with the classes it crosscuts according to specifications given within the aspect. Moreover, an aspect can use an *inter-type declaration* construct to declare fields, methods, and interface implementation declarations for classes. Declared members may be made visible to all classes and aspects (public inter-type declaration) or only within the aspect (private inter-type declaration), allowing one to avoid name conflicts with pre-existing elements.

A central concept in the composition of an aspect with other classes is called a *join point*. A join point is a well-defined point in the execution of a program, such as a call to a method, an access to an attribute, an object initialization, an exception handler, etc. Sets of join points may be represented by *pointcuts*, implying that such sets may crosscut the system. Pointcuts can be composed and new pointcut designators can be defined according to these combinations. AspectJ provides various pointcut *designators* that may be combined through logical operators to build up complete descriptions of pointcuts of interest [2].

An aspect can specify *advice* to define code that executes when a pointcut is reached. Advice is a method-like mechanism which consists of instructions that execute *before*, *after*, or *around* a pointcut. Around advice executes *in place* of the indicated pointcut, allowing a method to be replaced.

An AspectJ program can be divided into two parts: *base code* which includes classes, interfaces, and other standard Java constructs and *aspect code* which implements the crosscutting concerns in the program. Any AspectJ implementation must ensure that the aspect and base code run together in a properly coordinated fashion. The key component is the *aspect weaver*, which ensures that applicable advice runs at the appropriate join points.

Example. Figure 1 shows an AspectJ program taken from [2] that associates shadow points with every `Point` object. The program contains one aspect `PointShadowProtocol` and two classes `Point` and `Shadow`. The aspect has three methods `getShadowCount`, `associate` and `getShadow`, and three pieces of advice related to pointcuts `setting`,

```
public class Point {
  protected int x, y;
  public Point(int _x, int _y) {
    x = _x;
    y = _y;
  }
  public int getX() {
    return x;
  }
  public int getY() {
    return y;
  }
  public void setX(int _x) {
    x = _x;
  }
  public void setY(int _y) {
    y = _y;
  }
  public void printPosition() {
    System.out.println("Point
               at("+x+","+y+")");
  }
  public static void main(String[] args) {
    Point p = new Point(1,1);
    p.setX(2);
    p.setY(2);
  }
}
class Shadow {
  public static final int offset = 10;
  public int x, y;

  Shadow(int x, int y) {
    this.x = x;
    this.y = y;
  public void printPosition() {
    System.outprintln("Shadow at
          ("+x+","+y+")");
  }
}
```

```
aspect PS_Protocol {
  private int shadowCount = 0;
  public static int getCount() {
    return PS_Protocol.aspectOf().shadowCount;
  }
  private Shadow Point.shadow;
  public static void associate(Point p, Shadow s){
    p.shadow = s;
  }
  public static Shadow getShadow(Point p) {
    return p.shadow;
  }

  pointcut setting(int x, int y, Point p):
    args(x,y) && call(Point.new(int,int));
  pointcut settingX(Point p):
    target(p) && call(void Point.setX(int));
  pointcut settingY(Point p):
    target(p) && call(void Point.setY(int));

  after(int x, int y, Point p) returning :
    setting(x, y, p) {
    Shadow s = new Shadow(x,y);
    associate(p,s);
    shadowCount++;
  }
  after(Point p): settingX(p) {
    Shadow s = new getShadow(p);
    s.x = p.getX() + Shadow.offset;
    p.printPosition();
    s.printPosition();
  }
  after(Point p): settingY(p) {
    Shadow s = new getShadow(p);
    s.y = p.getY() + Shadow.offset;
    p.printPosition();
    s.printPosition();
  }
}
```

Figure 1: A sample AspectJ program.

`settingX` and `settingY` respectively[1]. The aspect also has two attributes `shadowCount` and `shadow` such that `shadowCount` is an attribute of the aspect itself and `shadow` is an attribute that is privately introduced to class `Point`. Through the rest of the paper, We use this example program to demonstrate our basic idea of coupling measurement for AO systems.

In the rest of the paper, we assume that an aspect is composed of attributes (aspect instance variables), and modules[2] such as advice, intertype declarations, pointcuts and methods.

[1]Unlike a method that has a unique method name, advice in AspectJ has no name. So for easy expression, we use the name of a pointcut to stand for the name of advice it associated with.

[2]For unification, we use the word a "module" to stand for a piece of advice, an intertype declaration, a pointcut, or a method declared in an aspect.

3 Terminology

In order to formally define our coupling measures in AO systems, we define a terminology for an AO system, which is based on a similar terminology used in [4] for object-oriented systems.

3.1 System

Our AO systems considered in this paper are composed of *aspects* and *classes*. We do not consider *interfaces* because they can be handled similarly with classes.

Definition 1 (AO System) *An AO system S consists of a set of aspects, $A(S)$ and a set of classes, $C(S)$.*

In an AO system, there may exist some inheritance relationships between aspects and/or classes. Here, we focus only on the inheritance relationships between aspects and classes.

Definition 2 (Ancestors of an Aspect) *Let S be an AO* *system. For each aspect $a \in A(S)$, let $Ancestors(a) \subset C(S)$ be the set of ancestor classes of a.*

3.2 Modules

In an AO system, an aspect contains several types of module, i.e., *advice*, *intertype declaration*, *pointcut*, and *method*, and a class contains only one type of module called *method*.

Definition 3 (Modules of an Aspect or a Class) *Let S be an AO system. For each $a \in A(S)$, let $\mathcal{A}(a)$ be the set of advice of a, $\mathcal{I}(a)$ be the set of intertype declarations of a, $\mathcal{P}(a)$ be the set of pointcuts of a, and $\mathcal{M}(a)$ be the set of methods of a, and $\mathcal{M}_{all}(a)$ be the set of all modules of a. For each $c \in C(S)$, let $\mathcal{M}(c)$ be the set of methods of c.*

In an AO system, advice, intertype declaration, pointcut, or method may have a set of parameters that may also influence coupling measurement. So we define this issue as follows.

Definition 4 (Parameters) *Let S be an AO system. For each $\alpha \in \mathcal{A}(S)$, $i \in \mathcal{I}(S)$, $p \in \mathcal{P}(S)$, or $m \in \mathcal{M}(S)$, let $Par(\alpha)$ be the parameters of advice α, $Par(i)$ be the parameters of intertype declaration i, $Par(p)$ be the parameters of pointcut p, and $Par(m)$ be the parameters of method m.*

3.3 Module Invocations

In AO systems, modules such as advice, intertype declarations, and methods in an aspect may invoke other modules of some classes. To measure coupling of an aspect, it is necessary to define the set of modules that a piece of advice, an intertype declaration, or a method of the aspect invokes. Also we should define the frequency of these invocations.

Definition 5 (The Set of Invoked Methods) *Let S be an AO system, $a \in A(S)$ be an aspect of S, and $c \in C(S)$ be a class of S.*

- *For each piece of advice $\alpha \in \mathcal{A}(a)$, the set of invoked methods of α is denoted as $SIM(\alpha)$ such that if $\exists m \in \mathcal{M}(c)$ and the body of α has a method invokation where m is invoked for an object of c, then $m \in SIM(\alpha)$.*
- *For each intertype declaration $i \in \mathcal{I}(a)$, the set of invoked methods of i is denoted as $SIM(i)$ such that if $\exists m \in \mathcal{M}(c)$ and the body of i has a method invokation where m is invoked for an object of c, then $m \in SIM(i)$.*
- *For each pointcut $p \in \mathcal{P}(a)$, the set of invoked methods of p is denoted as $SIM(p)$ such that if $\exists m \in \mathcal{M}(c)$ and the body of p has a method invokation where m is invoked for an object of c, then $m \in SIM(p)$. p*
- *For each method $m \in \mathcal{A}(a)$, the set of invoked methods of m is denoted as $SIM(m)$ such that if $\exists m' \in \mathcal{M}(c)$ and the body of m has a method invocation where m' is invoked for an object of c, then $m' \in SIM(m)$.*

Definition 6 (The Number of Method Invocations) *Let S be an AO system, $a \in A(S)$ be an aspect of S, and $c \in C(S)$ be a class of S.*

- *For each piece of advice $\alpha \in \mathcal{A}(a)$, $NSI(\alpha, m)$ is the number of method invocations of m by α such that $m \in SIM(\alpha)$ and m is invoked for an object of c.*
- *For each intertype declaration $i \in \mathcal{I}(a)$, $NSI(i, m)$ is the number of method invocations of m by i such that $m \in SIM(i)$ and m is invoked for an object of c.*
- *For each pointcut $p \in \mathcal{A}(a)$, $NSI(p, m)$ is the number of method invocations of m by p such that $m \in SIM(p)$ and m is invoked for an object of c.*
- *For each method $m' \in \mathcal{A}(a)$, $NSI(m', m)$ is the number of method invocations of m by m' such that $m \in SIM(m')$ and m is invoked for an object of c.*

3.4 Attributes

In AO systems, aspects and classes contain attributes that are either inherited or newly defined. Attributes are formally defined as follows:

Definition 7 (Attributes of Aspects and Classes) *Let S be an AO system. For each $a \in A(S)$, let $\mathcal{A}^a(a)$ be the set of attributes of aspect a. For each $c \in C(S)$, let $\mathcal{A}^a(c)$ be the set of attributes of class c.*

3.5 Types

Attributes and parameters contains types that all can contribute to coupling measurement of AO systems. The AO programming languages provide built-in types and support users to define new aspect and class types as well as traditional types. In AspectJ, the type of an attribute or parameter either is an aspect, a class, a built-in type or a user-defined type. Therefore, the set T of available types in an AO system is defined as follows:

Definition 8 (Available Types) *Let S be an AO system. The set T of available types in S is $T = T_{bi} \cup T_{ud} \cup T_c$ where T_{bi} is the set of built-in types provided by the programming language, T_{ud} is the set of user-defined types, and T_c is the set of class types.*

We next denote the type of attributes and parameters.

Definition 9 (Types of Attributes and Parameters) *Let S be an AO system, $x \in \mathcal{A}^a(a)$ be an attribute of aspect a, and $y \in \mathcal{A}^a(c)$ be an attribute of class c. The type of x is denoted by $T(x) \in T$ and the type of y is denoted by $T(y) \in T$.*

4 Coupling Framework for Aspect-Oriented Systems

We next present our coupling framework for AO systems which is a basis for defining coupling measures for AO system. Our framework focuses on coupling caused by dependencies that occur between aspect and class in an AO system, and therefore can handle issues of coupling measurement that are specific to AO systems.

Dependence between aspects and classes is a key point to determine the mechanism by which an aspect and a class are coupled. Here, we identify different types of dependencies that can be used to determine if a particular type more accurately indicates fault likelihood. In the following, we describe these types of dependencies between aspects and classes, which we call *aspect-class dependencies*. To define coupling measures in AO systems in Section 5, we also give the formal expressions for these dependencies.

Definition 10 (Attribute-class dependence) *There is an attribute-class dependence between aspect a and class c, if c is the type of an attribute of a. The number of attribute-class dependencies from a to c can formally be represented as*

$$AtC(a,c) = \left|\{\, x \mid x \in \mathcal{A}^a(a) \wedge T(x) = c\}\right|$$

Definition 11 (Module-class Dependence) *Let S be an AO system, $a \in A(S)$ be an aspect of S, and $c \in C(S)$ be a class of S. There are four types of module-class dependencies that can be defined as follows:*

- *Advice-class dependence:*
  There is an advice-class dependence between a and c, if c is the type of a parameter of a piece of advice α of a, or c is the return type of α. The number of advice-class dependencies from a to c can formally be represented as

$$AC(a,c) = \sum_{\alpha \in \mathcal{A}(a)} \left|\{\, x \mid x \in Par(\alpha) \wedge T(x) = c\}\right|$$

- *Intertype-class dependence:*
  There is an intertype-class dependence between a and c, if c is the type of a parameter of an intertype declaration i of a, or c is the return type of i. The number of intertype-class dependencies from a to c can formally be represented as

$$IC(a,c) = \sum_{i \in \mathcal{I}(a)} \left|\{\, x \mid x \in Par(i) \wedge T(x) = c\}\right|$$

- *Method-class dependence:*
  There is a method-class dependence between a and c, if c is the type of a parameter of a method m of a, or c is the return type of m. The number of method-class dependencies from a to c can formally be represented as

$$MC(a,c) = \sum_{m \in \mathcal{M}(a)} \left|\{\, x \mid x \in Par(m) \wedge T(x) = c\}\right|$$

- *Pointcut-class dependence:*
  Let p be a pointcut of aspect a. There is a pointcut-class dependence between a and c, if c is the type of a parameter of a pointcut p of a. The number of pointcut-class dependencies from a to c can formally be represented as

$$PC(a,c) = \sum_{p \in \mathcal{P}(a)} \left|\{\, x \mid x \in Par(p) \wedge T(x) = c\}\right|$$

Definition 12 (Module-method Dependence) *Let S be an AO system, $a \in A(S)$ be an aspect of S, and $c \in C(S)$ be a class of S. There are four types of module-method dependencies between a and c that can be defined as follows:*

- *Advice-method dependence:*
  There is an advice-method dependence between a and c, if a piece of advice α of a directly invokes a method m of c. The number of advice-method dependencies from a to c can formally be represented as

$$AM(a,c) = \sum_{\alpha \in \mathcal{A}(a)} \sum_{m \in M(c)} \left|(NSI(\alpha, m)\right|$$

- *Intertype-method dependence:*
  There is an intertype-method dependence between a and c, if an intertype i of a directly invokes a method m of c. The number of intertype-method dependencies from a to c can formally be represented as

$$IM(a,c) = \sum_{i \in \mathcal{I}(a)} \sum_{m \in M(c)} \left|(NSI(i, m)\right|$$

- *Method-method dependence:*
  There is a method-method dependence between a and c, if a method m of a directly invokes a method m' of c. The number of method-method dependencies from a to c can formally be represented as

$$MM(a,c) = \sum_{m \in \mathcal{M}(a)} \sum_{m' \in M(c)} \left|(NSI(m, m')\right|$$

- *Pointcut-method dependence:*
  There is a pointcut-method dependence between a and c, if a pointcut p of a contains at least one join point that is related to a method m of c. The number of pointcut-method dependencies from a to c can formally be represented as

$$PM(a,c) = \sum_{p \in \mathcal{P}(a)} \sum_{m \in M(c)} \left|(NSI(p, m)\right|$$

Definition 13 (Aspect-inheritance dependence) *There is an aspect-inheritance dependence between aspect a and class c, if c is an ancestor of a. The number of aspect-inheritance dependencies from a to c can formally be represented as*

$$AI(a,c) = \left| \{ \; x \mid x \in Ancestors(a) \} \right|$$

Example 1 *For the example program showed in Figure 1, Let a be apsect* `PS_Protocol` *and c be class* `Point`*, we can obtain the following dependencies between a and c.*

- $AtC(a,c) = 0$, $AC(a,c) = 3$, $IC(a,c) = 0$
- $MC(a,c) = 2$, $PC(a,c) = 3$, $AM(a,c) = 9$
- $IM(a,c) = 0$, $MM(a,c) = 0$, $PM(a,c) = 3$

5 Coupling Measures for Aspect-Oriented Systems

We next define our coupling measure suite that implements the coupling framework presented in Section 4. Our coupling measures are defined on counting, for each aspect a, the number of dependencies between a and some classes, i.e., *attribute-class*, *module-class*, *module-method*, and *aspect-inheritance* dependencies.

Definition 14 (Attribute-Class dependence Measure) *Let S be an AO system, $a \in A(S)$ be an aspect of S, and $c \in C(S)$ be a class of S. Attribute-class dependence measure can be defined as follows:*

$$\delta_{AtC}(a) = \sum_{c \in C(S)} AtC(a,c)$$

counts all AtC-dependencies between aspect a and some classes in S.

Definition 15 (Module-Class dependence Measure) *Let S be an AO system, $a \in A(S)$ be an aspect of S, and $c \in C(S)$ be a class of S. There are four types of module-class dependence based coupling measures that can be defined as follows:*

- *Advice-class dependence measure can be defined as follows:*

  $$\delta_{AC}(a) = \sum_{c \in C(S)} AC(a,c)$$

  counts all advice-class dependencies between aspect a and some classes in S.

- *Intertype-class dependence measure can be defined as follows:*

  $$\delta_{IC}(a) = \sum_{c \in C(S)} IC(a,c)$$

  counts all intertype-class dependencies between aspect a and some classes in S.

- *Method-class dependence measure can be defined as follows:*

  $$\delta_{MC}(a) = \sum_{c \in C(S)} MC(a,c)$$

  counts all method-class dependencies between aspect a and some classes in S.

- *Pointcut-class dependence measure can be defined as follows:*

  $$\delta_{PC}(a) = \sum_{c \in C(S)} PC(a,c)$$

  counts all pointcut-class dependencies between aspect a and some classes in S.

Definition 16 (Module-Method dependence Measure) *Let S be an AO system, $a \in A(S)$ be an aspect of S, and $c \in C(S)$ be a class of S. There are four types of module-method dependence based coupling measures can be defined as follows:*

- *Advice-method dependence measure can be defined as follows:*

  $$\delta_{AM}(a) = \sum_{c \in C(S)} AM(a,c)$$

  counts all advice-method dependencies between aspect a and some classes in S.

- *Intertype-method dependence measure can be defined as follows:*

  $$\delta_{IM}(a) = \sum_{c \in C(S)} AM(a,c)$$

  counts all intertype-method dependencies between aspect a and some classes in S.

- *Method-method dependence measure can be defined as follows:*

  $$\delta_{MM}(a) = \sum_{c \in C(S)} MM(a,c)$$

  counts all method-method dependencies between aspect a and some classes in S.

- *Pointcut-method dependence measure can be defined as follows:*

  $$\delta_{PM}(a) = \sum_{c \in C(S)} PM(a,c)$$

  counts all pointcut-method dependencies between aspect a and some classes in S.

Definition 17 (Aspect-Inheritance dependence Measure) *Let S be an AO system, $a \in A(S)$ be an aspect of S, and $c \in C(S)$ be a class of S. Aspect-inheritance dependence measure can be defined as follows:*

$$\delta_{AI}(a) = \sum_{c \in C(S)} AI(a, c)$$

counts all AI-dependencies between aspect a and all ancestor classes of a.

Example 2 *For the example program showed in Figure 1, Let a = aspect* `PS_Protocol` *and c = class* `Point`*, we can obtain the following results for some coupling measures defined in this section based on the results in Example 1.*

- $\delta_{AtC}(a, c) = 0$, $\delta_{AC}(a, c) = 3$, $\delta_{IC}(a, c) = 0$
- $\delta_{MC}(a, c) = 2$, $\delta_{PC}(a, c) = 3$, $\delta_{AM}(a, c) = 9$
- $\delta_{IM}(a, c) = 0$, $\delta_{MM}(a, c) = 0$, $\delta_{PM}(a, c) = 3$

6 Mathematical Properties of Coupling Measures

We next study the mathematical properties of our coupling measures proposed in Section 5. In particular, we look at five mathematical properties proposed by Briand et al.[4] for coupling measures of object-oriented systems. Although Briand et al. study these properties from the viewpoint of object-orientation, we believe that these properties are also useful in studying coupling measures for AO systems. In the following, we refine these properties in order to adapt them to study the coupling measures for AO systems. Let *Coupling* be a candidate measure for coupling of an aspect a or an AO system S. We define $Dep(a)$ to be the set of all dependencies between a and some classes in S, and $Dep(S)$ to be the set of all dependencies between aspects and classes in system S. We have $Dep(S) = \sum_{a \in A(S)} Dep(a)$.

6.1 Property Definitions

Property 1 (Nonnegativity) *Let S be an AO system, $a \in A(S)$ be an aspect of S.*

- *The coupling of S is nonnegative, i.e., $Coupling(S) \geq 0$.*
- *The coupling of a is nonnegative, i.e., $Coupling(a) \geq 0$.*

Property 2 (Null value) *Let S be an AO system, $a \in A(S)$ be an aspect of S.*

- *The coupling of S is null if $Dep(S)$ is empty, i.e., $Dep(S) = \phi \Rightarrow Coupling(S) = 0$.*
- *The coupling of a is null if $Dep(a)$ is empty, i.e., $Dep(a) = \phi \Rightarrow Coupling(a) = 0$.*

Property 3 (Monotonicity) *Let S be an AO system and $a \in A(S)$ be an aspect of S. We modify a to form a new aspect a' which is identical to a except that $Dep(a) \in Dep(a')$, i.e., we add some relationships to a. Let S_a be the AO system which is identical to S except that aspect a is replaced by a'. Then*

- *$Coupling(a) \leq Coupling(a')$, and*
- *$Coupling(S) \leq Coupling(S_a)$.*

Property 3 states that if a relationship between an aspect and a class is added to an AO system, coupling of the system must not decrease.

Property 4 (Merging of aspects) *Let S be an AO system, $a_1, a_2 \in A(S)$ be two aspects of S. Let a' be the aspect which is the union of a_1 and a_2. Let S_a be the AO system which is identical to S except that a_1 and a_2 are replaced by a'. Then*

- *$Coupling(a_1) + Coupling(a_2) \geq Coupling(a')$, and*
- *$Coupling(S) \geq Coupling(S_a)$.*

Property 4 states that merging of two aspects must not increase coupling because relationships disappear (namely those between the aspects that have been merged).

Property 5 (Merging of unconnected aspects) *Let S be an AO system, $a_1, a_2 \in A(S)$ be two aspects of S. Let a' be the aspect which is the union of a_1 and a_2. Let S_a be the AO system which is identical to S except that a_1 and a_2 are replaced by a'. If no relationships exist between a_1 and a_2 in S, then*

- *$Coupling(a_1) + Coupling(a_2) = Coupling(a')$, and*
- *$Coupling(S) = Coupling(S_a)$.*

Property 5 states that merging of two aspects that are not connected must not affect coupling at all.

6.2 Discussion and Comparison of Measures

For Properties 1, 2, and 3, all of our measures are satisfied. For properties 4 and 5, however, we should give some explanations here. Since our coupling measures proposed in this paper are only focused on dependencies between aspects and classes, it seems that the properties 4 and 5 described in [4] should be refined or re-defined as "merging of an aspect and a class" and "merging of an unconnected aspect and class". However, since merging of an aspect and a class is not valid for an AO system, we can not do so for

Table 1: Properties of Coupling Measures for AO Systems

Measures	**Type of coupling**	**Strength of coupling**	**inheritance**	**P1**	**P2**	**P3**	**P4**	**P5**
$\delta_{AtC}(a)$	type of attributes	#attributes	non-inh.-based	Yes	Yes	Yes	Yes	Yes
$\delta_{AC}(a)$	type of parameters	#parameters	non-inh.-based	Yes	Yes	Yes	Yes	Yes
$\delta_{IC}(a)$	type of parameters	#parameters	non-inh.-based	Yes	Yes	Yes	Yes	Yes
$\delta_{MC}(a)$	type of parameters	#parameters	non-inh.-based	Yes	Yes	Yes	Yes	Yes
$\delta_{PC}(a)$	type of parameters	#parameters	non-inh.-based	Yes	Yes	Yes	Yes	Yes
$\delta_{AM}(a)$	method call	#method calls	non-inh.-based	Yes	Yes	Yes	Yes	Yes
$\delta_{IM}(a)$	method call	#method calls	non-inh.-based	Yes	Yes	Yes	Yes	Yes
$\delta_{MM}(a)$	method call	#method calls	non-inh.-based	Yes	Yes	Yes	Yes	Yes
$\delta_{PM}(a)$	method call	#method calls	non-inh.-based	Yes	Yes	Yes	Yes	Yes
$\delta_{AI}(a)$	inheritance	#ancesters	inh.-based	Yes	Yes	Yes	Yes	Yes

properties 4 and 5. However, since it is really meaningful if we merge two aspects in an AO system, we can refine properties 4 and 5 as we showed in this section.

Table 1 summarizes the discussion of our measures. For each measure, we show the type of coupling it uses, the factors that determines the strength of coupling, how inheritance is dealt with (inheritance-based coupling, non-inheritance-based coupling, or both). The columns P1, P2, P3, P4, and P5 show whether or not a measure satisfies these five properties.

7 Related work

We discuss some related work in the area of coupling measurement for object-oriented systems and also measuring the complexity of AO systems. To the best of our knowledge, our work is the first attempt to study the coupling measurement for AO systems.

7.1 Coupling Measurement in Object-Oriented Systems

Chidamber and Kemerer [7] propose a coupling measure called the *Coupling Between Object* (CBO) classes to assess class coupling in object-oriented systems. With the CBO measures, class C_1 is coupled to class C_2 if C_1 uses C_2's member functions and/or instance variables. CBO counts the number of classes to which a given class is coupled. Briand et al. [5] investigate the coupling measures for C++ programs. They propose a comprehensive suite of measures to quantify the level of class coupling during the design of object-oriented systems. Their coupling measures are defined based on various types of interactions between classes in an object-oriented system. Briand et al [4] also present a unified framework for comparing coupling measures for object-oriented systems from various viewpoints. Roughly speaking, our work can be regarded as an extension of the work by Briand *et.al* [5, 4] to handle the unique problems of coupling measurement for AO systems. Coupling measurement for object-oriented systems has also been studied by Li and Harry [11] who propose some coupling measure in terms of massage passing and data abstraction for object-oriented systems and Martin [15] who proposes efferent and afferent coupling for object-oriented programs. Moreover, Hitz and Montazeri [10] present a framework for measuring class-level and object-level coupling for object-oriented systems.

Although these coupling measures can be used to assess the coupling for object-oriented systems from different viewpoints, they can not be directly applied to AO systems since these approaches only consider the interactions between classes in object-oriented systems. Since an AO system contains aspects that are significantly different from classes in object-oriented systems, new abstraction models are needed for representing various types of interactions between aspects and classes in order to derive new coupling measures for AO systems.

7.2 Measures for Aspect-Oriented Systems

Zhao [18] proposes a metrics suite for aspect-oriented software, which are specifically designed to quantify the information flows in an aspect-oriented program. To this end, He presents a dependence model for aspect-oriented software which is composed of several dependence graphs to explicitly represent dependence relationships in a module, a class, or the whole program. Based on the model, he defines some metrics that can be used to measure the complexity of an aspect-oriented program from various different viewpoints. Recently, cohesion measurement for AO systems has also been studied by Zhao and Xu [19]. However, these studies do not address the issue on coupling measurement for AO systems.

8 Concluding Remarks

In this paper, we proposed a measure suite for assessing the coupling in AO systems. We first presented a coupling framework for AO systems which specially designed to count the dependencies between aspects and classes in the system. Based on this framework, we formally defined various coupling measures in terms of different types of dependencies between aspects and classes. We also discussed the mathematical properties of these measures in which we showed that our measures satisfy the properties that a good coupling measure should have.

The coupling measures proposed in this paper focused only on the dependencies between aspects and classes, and did not take dependencies between aspects or between classes into account.

In our future work, we will study the influence of dependencies between aspects or classes, aspect and class inheritances, and other aspect-oriented features on coupling measurement of AO systems, and apply our coupling measures to real aspect-oriented system design.

References

[1] F. Abreu, M. Goulao, and R. Esteves, “Toward the Design Quality Evaluation of Object-Oriented Software Systems,” *Proc. 5th International Conference on Software Quality*, Austin, Texas, October 1995.

[2] The AspectJ Team. The AspectJ Programming Guide. 2003.

[3] L. Bergmans and M. Aksits. Composing crosscutting Concerns Using Composition Filters. *Communications of the ACM*, Vol.44, No.10, pp.51-57, October 2001.

[4] L. C. Briand, J. Daly and J. Wuest. A Unified Framework for Coupling Measurement in Object-Oriented Systems. IEEE Transactions on Software Engineering, Vol.25, No.1, pp.91-121, January/February 1999.

[5] L. C. Briand, P. Devanbu, and W. Melo, “An Investigation into Coupling Measures for C++,” *Proc. 19th International Conference on Software Engineering*, pp.412-421, May 1997.

[6] L. C. Briand, S. Morasca and V. Basili. Property-Based Software Engineering Measurement. *IEEE Transactions on Software Engineering*, Vol.22, No.1, pp.68-86, 1996.

[7] S. R. Chidamber and C. F. Kemerer. A Metrics Suite for Object-Oriented Design. *IEEE Transactions on Software Engineering*, pp.476-493, Vol.20, No.6, 1994.

[8] J. Eder, G. Kappel, and M. Schrefl. Coupling and Cohesion in Object-Oriented Systems. Technical Report, University of Klagenfurt, 1994.

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

[10] M. Hitz and B. Montazeri. Measuring Coupling and Cohesion in Object-Oriented Systems. *Proceedings of International Symposium on Applied Corporate Computing*, pp.25-27, Monterrey, Mexico, October 1995.

[11] W. Li and S. Henry. Object-Oriented Metrics that Predict Maintainability. *Journal of Systems and Software*, Vol.23, No.2, pp.111-222, 1993.

[12] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, and J. Irwin, “Aspect-Oriented Programming, “ *Proceedings of the 11th European Conference on Object-Oriented Programming*, pp.220-242, LNCS, Vol.1241, Springer-Verlag, June 1997.

[13] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, and J. Irwin, “An Overview of AspectJ,” *Proc. 13th European Conference on Object-Oriented Programming*, pp.220-242, LNCS, Vol.1241, Springer-Verlag, June 2000.

[14] K. Lieberher, D. Orleans, and J. Ovlinger, “Aspect-Oriented Programming with Adaptive Methods,” *Communications of the ACM*, Vol.44, No.10, pp.39-41, October 2001.

[15] R. Martin, “OO Design Quality Metrics - An Analysis of Dependencies,“ position paper, *Proc. Workshop Pragmatic and Theoretical Directions in Object-Oriented Software Metrics, OOPSLA’94*, October 94.

[16] L. M. Ott, J. M. Bieman, B. K. Kang, and B. Mehra. ”Developing Measures of Class Cohesion for Object-Oriented Software,” *Proc. 7th Annual Oregon Workshop on Software Metrics*, June 1995.

[17] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, “N Degrees of Separation: Multi-Dimensional Separation of Concerns,” *Proceedings of the International Conference on Software Engineering*, pp.107-119, 1999.

[18] J. Zhao, “Toward A Metrics Suite for Aspect-Oriented Software,” Technical Report SE-136-5, Information Processing Society of Japan (IPSJ), March 2002.

[19] J. Zhao and B. Xu, “Measuring Aspect Cohesion,” *Proc. Fundamental Approaches to Software Engineering (FASE’2004)*, LNCS 2984, pp.54-68, Springer-Verlag, Barcelona, Spain, March 2004.