Towards a Unified Coupling Framework for Measuring Aspect-Oriented Programs

Thiago T. Bartolomei
Dep. Computer Science and Electrical Engineering
Kiel University of Applied Sciences
thiago.bartolomei@gmail.com

Alessandro Garcia, Claudio Sant'Anna and Eduardo Figueiredo
Computing Department
Lancaster University
{garciaa, c.santanna, e.magno}@comp.lancs.ac.uk

ABSTRACT

There is nowadays a wide recognition that low coupling is a main tenet in the measurement of high-quality modular software. In fact, coupling is one of the few internal software attributes that has been both theoretically and empirically shown to have a large impact on a variety of external software qualities, such as reusability, maintainability, evolvability and testability. Aspect-oriented programming (AOP) is an emerging technique that advocates enhanced modularization of certain widely-scoped system properties, the so-called *crosscutting concerns*. However, there is a poor common understanding of coupling in the context of AOP. Most of the existing metrics and assessment frameworks concentrate on the coupling evaluation of the AspectJ programming language. In addition, they took their own particular view of what coupling means in this particular language. This paper presents the definition of a generic coupling framework that takes into account both AspectJ and CaesarJ, two representatives of the most well-known families of available AOP languages. The current version of the proposed framework allows for the definition of different coupling metrics, which in turn permits the analysis and comparison of Java, AspectJ and CaesarJ implementations. We also illustrate how the framework can be applied to the characterization of existing coupling metrics.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics; D.3.0 [**Programming Languages**]: General

General Terms

Measurement, Experimentation, Languages

Keywords

Metrics, coupling, aspect-oriented programming, empirical assessment

1. INTRODUCTION

With the emergence of aspect oriented programming (AOP) languages and its advanced composition mechanisms for promoting enhanced separation of *crosscutting concerns*, there is a pressing need to assess how the new forms of module connection in AOP impact stringent external qualities, such as reusability, maintainability, and testability. Remarkably the transfer of AOP technologies to mainstream software development is largely dependent on our ability to empirically understand its positive and negative effects on well-known quality design principles other than separation of concerns, such as low coupling. In fact, coupling measures are one of the few evaluation mechanisms that have been both theoretically and empirically validated with respect to their direct influence on various external quality attributes. As a result, it is essential to come up with a common understanding of coupling measures in aspect oriented (AO) systems.

However, the definition of a unified measurement framework for coupling is not a trivial task thanks to the heterogeneous variety of composition mechanisms supported by different AOP languages, such as AspectJ [13] and CaesarJ [2, 16]. For example, AspectJ is a seamless extension to Java that introduces new concepts, such as *aspect* and *advice*, and supports both static and dynamic composition features, such as *inter-type*, *pointcut*, and *precedence declarations*. Some existing AOP languages and frameworks provide a very similar composition model to the AspectJ one, such as Springs AOP framework [12] and JBoss AOP [11]. On the other extreme, CaesarJ 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 coupling in AO systems [7, 17, 18], there is no unified measurement framework for characterising what constitutes coupling in AO systems. Sant'Anna and colleagues [17, 10] have defined two coupling metrics for AO design and programming, and applied them to a number of empirical case studies (such as [9, 10, 6, 14]). In addition, other works, such as [7], define a different set of AO coupling metrics. However, all these measures: *(i)* capture only a restricted set of aspectual connection dimensions by extending the original metric definitions of Chidamber and Kemerer (CK) metrics [8], and *(ii)* are very specific to the composition model defined by the AspectJ language. As a result, they do not support the coupling comparison of similar alternatives implemented in different AOP languages.

Briand et al[5] introduced a framework for coupling measurements for OO systems. However, since AO systems introduce new abstractions that were unknown when this framework was created, it cannot be directly applied to AO measurements and extensions are needed. A first step in this direction was made by Bartsch and Harrison [4], who extended the framework for AspectJ. Nonetheless, their framework concentrate on AspectJ specific coupling connections and do not deeply discuss other important points, such as dynamic issues.

This paper reports our ongoing work on the description of a unified coupling framework that builds on top of existing coupling measurement approaches for AOP [17, 10, 18, 4, 7]. The current version of the proposed framework refines Briand's framework and has specifically targeted at the composition models supported by Java, AspectJ and CaesarJ. The goals of our framework are : *(i)* to create a standard terminology that provides the basic foundation for defining coupling measures in AO programs, *(ii)* to impose operational definitions for coupling measurements and *(iii)* to increase practitioners and researchers awareness of decisions that must be taken when developing new coupling metrics suites for these languages.

The paper is structured accordingly. Section 2 defines a terminology for fundamental structural elements in AO systems. After describing the framework in Section 3, we demonstrate its application by categorising an existing coupling metric for AOP in Section 4. In Section 5 we discuss our contribution in the light of related work. Conclusion remarks and future work are presented in Section 6.

2. TERMINOLOGY

In order to consistently and unambiguously define coupling measurements it is necessary to establish a proper terminology. Briand et al [5] introduced a terminology for object oriented (OO) systems based on set and graph theory, which Zhao [18] extended with some of the AO concepts supported by AspectJ-like languages. This section presents terminological foundation for structural elements in an AO system. The terminology encompasses the core abstractions across all the studied programming languages, namely Java, AspectJ, and CaesarJ. At the same time, the basic terminology is agnostic to the specificities of each of these languages and its composition mechanisms.

2.1 Structure

System, Components, and Members

We define an *Aspect-Oriented (AO) system*'s structure as a set of fundamental elements, which subsumes its key abstractions for module specifications. Figure 1 presents a detailed view of the structural abstractions defined for a system. Each type of abstraction is alternatively called an *element*.

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 $M(c) = Att(c) \cup Op(c) \cup Nested(c)$. The reader should notice that for generality purposes, a component is an unified 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, 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, $JP(cod)$.

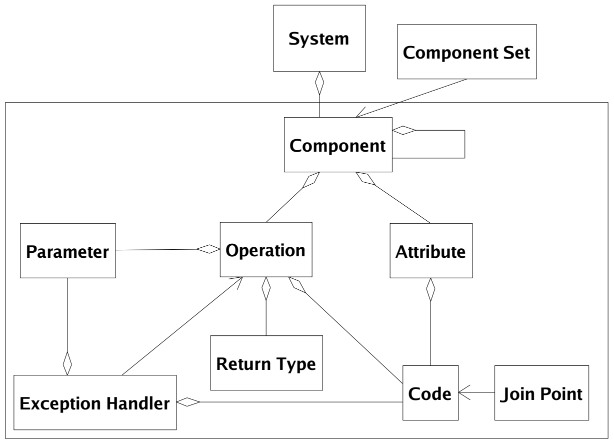

Figure 1: Structure of an Aspect-Oriented System

Components may participate on inheritance relationships. For a given component c, the following sets are defined: *(i)* $Ancestors(c)$ - all recursively defined parents; *(ii)* $Parents(c)$ - the directly declared parents; *(iii)* $Children(c)$ - the directly derived children, and *(iv)* $Descendants(c)$ - the recursively derived children. Because of 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

Coupling Connections, Views and Relationships

The definition of a *Coupling Connection* is a dependency relationship between two elements, where one element, called *Server*, provides a service to another element, the *Client* [5]. Component sets may be defined as a view over components. A component c may belong to a set of component sets, $Set(c)$. The components that belong to a set s are defined by $C(s)$. An exception handler may be seen as a view of an operation's code. The set of handlers of an operation o is defined as $H(o)$. An exception handler h contains code, $Code(h)$ and may have a set of parameters, $Par(h)$. If an element e consists of an element f, then e is said to be *Coarser* than f; f is *Finer* than e.

Types and Notational Convenience

The set of all available types of a system S is denoted by $T(S) = BT(S) \cup UDT(S) \cup CT(S)$, which is the join set of Built-In Types, User Defined Types and Component Types, respectively. The types of an attribute a and a parameter p are defined by $T(a)$ and $T(p)$, respectively.

For convenience purpose, we define that the set of all components of a system, $C(S)$, can be partitioned in 3 subsets: Application ($AC(S)$), Framework ($FC(S)$) and Library ($LC(S)$). The set of all components, attributes, operations, exception handlers, join points, parameters, return types and code of a system S, including nested components, is denoted by $AC(S)$, $A(S)$, $O(S)$, $H(S)$, $JP(S)$, $P(S)$, $Rt(S)$ and $CODE(S)$, respectively.

2.2 Behaviour

This section defines the required formalism for the behaviour of an AO system. The definitions are introduced below for the different kinds of operation invocation in AO systems, and polymorphic influences on attribute references.

Operations

Two types of operation invocations can be defined : *explicit invocation* and *implicit invocation*. Explicit invocation is caused by code calling the operation. Implicit invocations are caused by a join point being reached or by an exception handler catching an exception. Therefore, there is a list of operation invocation sets defined for an element e, which are extensively described and defined at [1]. The different types of operation invocations are jointly determined by the nature of the invocation i.e. implicit or explicit invocation and the strategy for considering ancestor elements. Some examples that will be further explored in this paper (Section 4) are: *(i)* $EOI_{ID}(e)$ - explicit operation invocations, accounting for implementing descendants of the operation's component; *(ii)* $IOI_{ST}(e)$ - implicit operation invocations that can be fully statically determined; and *(iii)* $IOI_{DYN}(e)$ - implicit operation invocations that can be only dynamically determined. These sets can be extended with a meta variable c' which represents the component that is the invocation target. Note that this component not always implements the invoked operation, due to inheritance relationships. The sets with the meta variable are defined as $EOI_{ST}(e, c')$, and so on.

Attributes

Attribute references may also be influenced by polymorphism. Hence, a complementary list of sets of attribute references are defined for an element e. The complete list is available at [1]. Due to space limitation, we just mention some examples here, such as: *(i)* $AR_{ST}(e)$ - attribute references, accounting only for statically determined attributes; *(ii)* $AR_{ID}(e)$ - attribute references, accounting for implementing descendants of the attribute's component; and *(iii)* $AR_{FI}(e)$ - attribute references, accounting only for the first statically determined implementation, looking for the attribute's component and its ancestors. Similarly to operation invocations, we define sets with a meta variable representing the target component: $AR_{ST}(e, c')$, and so on.

2.3 Language Instantiation

The aforementioned abstract structure can be instantiated for different languages. The following items below define how the structure can be mapped for Java, AspectJ(A) and CaesarJ(C). Abstractions that map to all languages are marked with an All. Note that aspects and inter-type declarations are only valid for AspectJ-like languages, while CClasses are defined only for CaesarJ.

Component Set Can be both a free selection of components, but also a Package(All).

Component Class (All), Interface (All), Aspect (A) and CClass (C).

Attribute Field (All) and Intertype Field (A).

Operation Method (All), Constructor (All), Static Initializer (All), Intertype Method (A), Intertype Constructor (A), Intertype Attribute (A), Pointcut (A, C), Advice (A, C), Declare Statements (A, C) and Wrapper Constructor(C).

3. THE COUPLING FRAMEWORK

In order to define a unified coupling framework, we have analysed frameworks for OO systems [5, 3], AO systems [18] and specific for AspectJ [4]. Moreover, we have identified the influences of CaesarJ and its special feature oriented decomposition mechanisms. From these analyses a list of criteria that must be considered when developing a new coupling measurement emerged. Each criterion is supposed to answer a question. We have tried to maximise the use of the terminology and criteria originally defined in Briand's frameworks [5, 3] whenever it was possible.

The following list presents our criteria and the questions each criterion is supposed to answer. The criteria is ordered according to the precedence they must have, because some decisions in a criterion may impose constraints to another criterion. The main impact of aspect-orientation was on the definition and extensions provided for the criteria related to domain, connection types, dynamic issues, and language-specific considerations. The next subsections provide details on each criterion.

1. **Granularity** - What elements to measure?
2. **Direction** - What is the locus of impact?
3. **Aggregation** - How to aggregate connections?
4. **Domain** - Which elements to account for, which to restrict?
5. **Connection Types** - What constitutes coupling?
6. **Dynamic Issues** - How to deal with dynamic aspects of the system?
7. **Language Specific Considerations** - Are there language specific couplings that were not modelled by the framework?

3.1 Granularity

What elements to measure?

This criterion determines the elements that are going to be measured, i.e., to which elements the measurements apply. Usually, coupling measurements have *Component* as the *granularity*, but other selections, such as *ComponentSet* or *System* also exist. As a result, the following elements can be considered: *(i)* System, *(ii)* Component Set, *(iii)* Component, *(iv)* Attribute, *(v)* Operation, and *(vi)* Exception Handler. Notice that the selection of granularity means that internal connections, between elements that compose the element selected for *granularity*, are excluded from the counting. For example, if we choose *Component* for the *granularity*, connections between elements (such as internal advice and pointcuts) of the same component are not counted as coupling. In the criterion *Dynamic Issues* of Section 3.6 we will discuss what really belongs to an element.

3.2 Direction

What is the locus of impact?

We have already shown in Section 2 that a coupling connection involves a server and a client. It must be decided if the elements are to be counted in their roles as servers or clients in this connection. The implications of this decision are exemplified in the next criterion. There are three possibilities: *(i)* **Import** - count elements in the role as clients of coupling connections, *(ii)* **Export** - count elements in the role as servers of coupling connections, and *(iii)* **Both directions** - count elements in both roles.

3.3 Aggregation

How to aggregate connections?

It must be decided how to aggregate connections and how to count them. There are four different facets of this decision. First, it must be decided in which elements connections are aggregated. Second, it must be decided which side of the connections to count, with respect to the *granularity* element. Third, if we count all connections or only distinct endpoints. Fourth, a level of transitivity must be defined. The transitivity means how many steps further the measure is recursively applied.

3.3.1 Aggregation Elements and Side

The *aggregation elements* define what is being counted, i.e. in which elements to aggregate connections. Possible values are all the structural elements. This decision depends on the *granularity* element, since the element chosen here must be a *Finer* element. For example, when we say "*the number of methods of a class that...*", the *granularity* is *Component* (class) but what we are counting is *Operation* (method).

The *aggregation side* defines which side of the connections are counted, with respect to the *granularity* element. We can choose to count elements in the same side as the *granularity* element or in the opposite side. For example, *DAC* [15] counts the number of attributes of a component, i.e., counts the same side as the component. DAC' counts the number of components that are the types of these attributed. Thus, DAC' counts the opposite side of the component.

3.3.2 Distinct Elements and Transitivity

Distinct elements count individual connections or only distinct endpoints. Distinct elements define if having several connections between the *granularity* element and another element will count for only one or for the number of connections. For example, *CBO* [8] counts only distinct elements that are coupled to the *granularity* element. Thus, having two connections between the components is the same as having only one.

Transitivity relates to how many steps to recursively apply the metric. Possible values are integers, zero meaning no transitivity. *RFC* [8], for instance, uses a level of transitivity of 1, i.e., the metric is applied to an *Operation* and then applied to all endpoints of the connections. RFC_α counts for α steps of transitivity.

3.4 Domain

Which elements to account for, which to restrict?

In *granularity* it was selected what is being measured. In *aggregation* it was selected what and how to count. In *domain* it is selected which elements are allowed to participate on coupling connections as servers or clients. There are 3 pertinent issues of this selection : how to account for inheritance; how to restrict types of elements; how to restrict elements based on component classification. In inheritance it is defined if components related via inheritance should be excluded or included in the coupling measurements. For example, *CBO* does not restrict the measure between inherited members, i.e., coupling between a component and the parents is also accounted. However, CBO' ignores this connection.

It is also needed to select the types of elements that should be accounted for. For example, Zhao's measurements [18] count only connections between aspects and classes. Since they are all *import* measures, the *Client Domain* is *Aspect* and the *Server Domain* is *Class*. It is also possible to select finer grained elements. For example, Zhao's *AM* measure counts only connections between *Advice* and *Classes*. Hence, *Advice* is the *Client Domain*. Notice that this selection is language dependent and very low level. For instance, we may select only singleton aspects as a domain.

The last issue is the classification of the server and client components. We defined in the terminology that components can be separated into application, framework and library components. Thus, we may restrict the client and server domains to only some types of this components. In short, for both client and server domains we may restrict elements based on: *(i)* **Inheritance** - Ancestors, Descendants, etc.; *(ii)* **Element Type** - Class, Aspect, Advice, Singleton Aspect, etc.; and *(iii)* **Component Classification** - Application, Framework, Standard Libraries.

3.5 Connection Types

What constitutes coupling?

This criterion defines exactly what causes coupling, i.e., which are the types of connections between elements that are relevant for the measure. In this framework we take into account connections related to OO coupling [5], AspectJ-specific connections [4] and generic AO connections [18]. Moreover, we studied possible coupling connections caused by CaesarJ-specific constructs. There are several possible classification criteria to separate the set of connections in relevant groups. We present here the set of coupling connections classified by *Nature of Composition*, because we believe it is a relevant classification criteria that may represent potential external measurement goals.

Our abstraction of coupling connection is a client-server relationship between structural elements. If we see coupling connections as a result of element compositions, *Nature of Composition* classifies connections based on characteristics of this composition. This classification groups connections according to 2 criteria: *OO vs AO* and *Structural vs Behavioural*. We briefly discuss each criteria and why we believe this classification is relevant. In the following subsections we describe in more detail each type of connection that was identified in the studied languages. The connections are classified according to our criteria.

OO vs AO - it seems important to separate coupling caused by OO compositions from AO compositions, because we may want to create different measurements to understand the influences of the different compositions. Thus, we separate connections that are present in OO systems from the ones that are introduced by AO mechanisms.

Structural vs Behavioural - this distinction is important because we may want to create measurements to understand coupling caused only by the structural composition of elements, but may also want to measure only coupling caused by the behavioural composition. Structural connections are created by type references in declarations, while behavioural connections are created by operation invocations, attribute references and statements. Notice that even in structural connections we still have a dynamic issue. For example, when there is coupling caused by a type reference to an interface, the dynamic type of the object implementing it is not statically known. This issue is discussed in the next criterion.

3.5.1 Object-Oriented Structural Connections

Component Declarations types in direct inheritance declarations.

Attribute Declarations types of attribute and local variable declarations.

Operation Declarations types in parameters of operation declarations and exception handlers; return types in operation declarations (including throws clauses).

3.5.2 Object-Oriented Behavioural Connections

Operation Invocations targets of invoked operations and types of their parameters and return types.

Attribute References targets of attribute references and types of referenced attributes.

Statements raising exceptions, array creation and type casts.

3.5.3 Aspect-Oriented Structural Connections

Component Declarations types in binding declarations.

Attribute Declarations types and targets in intertype attribute declarations.

Operation Declarations types in parameters of pointcuts, advice and intertype operation declarations; targets in intertype operation declarations; return types in advice and intertype operation declarations; explicit naming in pointcut declarations.

Declare Statements types in declare statements.

Listing 1: Member Definitions

```
public cclass A {
  public void a(C c) {}
}
public cclass B extends A {
  public void b() {}
}
public cclass C {}
```

3.5.4 Aspect-Oriented Behavioural Connections

Operation Invocations targets of join point implicit invocations and wrapper constructor invocations; targets of invoked intertype operations, types of their parameters and return types.

Attribute References targets of wrappee references and types of referenced wrappees; targets of intertype attribute references and types of referenced intertype attributes.

3.6 Dynamics Issues

How to deal with dynamic aspects of the system?

This coupling framework deals mostly with static aspects of the system. Dynamic issues, although outside the scope of a static measure, influence the measurements. Therefore, we must take decisions regarding how to arbitrarily deal with these issues.

3.6.1 Which members belong to a component

In order to measure coupling of a component, we have to decide what are the members of this component that influence the measurement. We have to create a set that is the union of some of these sets:

1. Members newly implemented in the component (not overridden)
2. Members declared as virtual in the component
3. Members inherited and overridden
4. Members inherited and not-overridden

These sets are disjoint and any combination is possible. If the measurement is more interested in static aspects of coupling, then a set such as $\{1, 2, 3\}$ should be defined, because it selects the members whose code can be found directly in the component. The set $\{1, 2, 3, 4\}$ deals additionally with dynamically assigned members. For example, Listing 1 defines three CClasses in CaesarJ. A is statically coupled to C due to the parameter in its method $a()$, but B has no mention of C in its code. According to our first set, B is not coupled to C. But dynamically, an object of type B will have the method $a()$ and using our second set we can measure this coupling to C.

3.6.2 Polymorphism

It must be decided how to determine the possible components in the endpoints of connections to members, when polymorphism is involved. For this criterion, we extend the options defined in [5], including the last 4 options.

Listing 2: Inheritance Relationships in CaesarJ

```
public cclass A {
  public void m() {}
}
public cclass B {}
public cclass C extends B & A {
   public void m() {}
}
public cclass D extends C {}

public class Client {
  A a = new A(); C c = new C();
  D d = new D();

  public void method() {
    a.m() // (i)
    c.m() // (ii)
    d.m() // (iii)
  }
}
```

1. Do not account for polymorphism (Static - ST) - only the connection with the statically determined elements (*static element*) is taken into account.

2. Account for implementing descendants (Implementing Descendants - ID) - connections with all descendants of the static element, that override the member, are taken into account.

3. Account for potential descendants (Potential Descendants - PD) - connections with all descendants of the static element are taken into account.

4. Account for potential ancestors (Potential Ancestors - PA) - connections with all ancestors of the static element are taken into account.

5. Account for implementing ancestors (Implementing Ancestors - IA) - connections with all ancestors of the static element, that implement the member, are taken into account.

6. Account for first implementing element (First Implementing - FI) - only the connection with the first implementing element is taken into account.

ST defines that only the static knowledge causes coupling. The relevance of ID is that it takes into account all possible implementations of the element that could cause the connection. PD counts all possible places where new implementations could cause the connection. The difference is that PD takes into account coupling due to potential changes. Listing 2 presents some CClasses implemented in CaesarJ. If counted as ID, the method call i, would have connections A and C, because these are the methods that would possibly be called at runtime. If counted as PD, it would have connections to A, C and D, because D can potentially implement this method.

PA looks to all ancestors of the component. The idea is that all these members potentially affect the connection. For example, a call to a constructor in Java causes a call in each ancestor's constructors. IA takes only members that are currently implemented, that is, it does not take potential implementations into account. With FI it is possible to account only for the first statically determined implementation. In our example, if method call iii is counted as PA, than there would be connections to A, B, C and D, because they all potentially could have an implementation of this method. If counted as IA, only connections with A and C would be counted. For FI there is only a connection with C, because it is the first ancestor of D to implement the method.

Table 1 presents for which components there would be connections for the method calls, according to the polymorphism option. It is important to notice that polymorphism affects not only operation explicit invocations (method calls). In CaesarJ, nested components are virtual classes and may be overridden. Thus, attribute references are also polymorphically defined.

Call	ST	ID	PD	PA	IA	FI
i	A	A,C	A,C,D	A	A	A
ii	C	C	C,D	A,B,C	A,C	C
iii	D	D	D	A,B,C,D	A,C	C

Table 1: Coupling due to the method calls

3.6.3 Dynamic Pointcuts

Most of the connections related to join points can be statically inferred, because the pointcuts determine exactly which join points are picked up. However, some pointcuts refer to dynamic states and it is only possible to determine if the join point is going to be picked up at runtime. In AspectJ and CaesarJ, the following pointcuts refer to dynamic properties: $cflow$, $cflowbelow$, $this$, $target$, $args$ and if (except for $if(true)$ and $if(false)$).

The question is how to deal with such pointcuts. When these pointcuts are used we still match some possible join points. These join points are potentially matched, but we need a runtime test to check whether they really match or not. Thus, we have two options:

1. consider connections involving join points matched by dynamic pointcuts

2. discard connections involving join points matched by dynamic pointcuts

The implication of this decision is the following. If we choose 1, we may actually be counting coupling where it does not really exist at runtime. Our measure will be then a *ceiling measure*. If we choose 2 we will be discarding connections and our measure represents a *floor measure*.

3.7 Language Specific Considerations

Are there language specific couplings that were not modelled by the framework?

Although the framework abstracts language specific instantiations, there may be some cases that were not foreseen. Thus, when a specific language is targeted, it is important to analyse if there are some specific aspects of the language that may contribute to coupling.

4. FRAMEWORK APPLICATION

In this section we demonstrate the application of our coupling framerwork by describing an existing coupling metric. First, we describe the metric textually. Second, we go through the whole process of analysing and selecting the criteria. Finally, we derive a formal definition.

4.1 Response for a Module (RFM)

RFM is a coupling metric proposed by Ceccato et al[7]. It is an AO extension of the OO *Response for a Class* (RFC) metric defined by Chidamber and Kemerer[8]. RFC is defined for a class as the set of methods that can be executed in response to a message received by an object of that class. To extend the RFC metric, Ceccato considers that a module can be a class or an aspect and defines RFM as the set of methods and advices potentially executed in response to a message received by a given module. Therefore, based on our framework, RFM is the response set of operations for a component.

4.2 Criteria Selection

We have classified the RFM metric according to the options available for each criterion of the framework. We briefly describe each criterion selection.

1. Granularity
The RFM definition clearly states that *Component* is the *granularity*, i.e., the measurement information is gathered at the component level.

2. Direction
Components are analysed in their roles as clients, because RFM counts the number of operations which are explicitly or implicitly invoked. Therefore, RFM measures the *Import Coupling* of a component.

3. Aggregation
Since RFM counts the number of potentially invoked operations, *aggregation elements* are the *Operations*. For a certain component that invokes operations, the operations are counted. Hence, the *aggregation* side is the *opposite side* of the granularity element. Moreover, only *distinct operations* are counted and the *transitivity* is 1.

4. Domain
Both *client domain* and *server domain* are *Operations* and no other types of element restrictions are necessary.

5. Connection Types
RFM extends RFC by also considering implicit invocations that are triggered whenever a join point is reached during the execution of an operation of a component. Hence, the *types of connections* are *targets of invoked operations* (explicit invocation) and *targets of join points implicit invocations.*

6. Dynamic Issues
RFM is a metric interested in the dynamic coupling. Hence, it selects all options of *member assignment*, with the set $\{1, 2, 3, 4\}$. In terms of *polymorphism*, it takes into account all implementing descendants of the static element in the endpoint of the connection (ID). In addition, RFM *considers connections involving join points matched by dynamic pointcuts.*

7. Language Specific Considerations
This metric is not involved in any language specific issue.

4.3 Formal Definition

Using the selected criteria and the AO terminology described in Section 2 we derive the following formal definition for RFM:

$$RFM_\alpha(c) = \begin{cases} 0 & if \quad c \in FC(S) \mid c \in LC(S) \\ & or \quad \mid \bigcup_{i=0}^{\alpha} R_i(c) \mid ; \alpha = 1, 2, 3... \end{cases}$$

where $R_0(c) = O(c)$ and
$R_{i+1} = \bigcup_{o \in R_i(c)} (EOI_{ID}(o, c') \cup IOI_{ST}(o, c') \cup IOI_{DYN}(o, c')) \mid c' \in (C(S) - (\{c\} \cap Ancestors(c)))$

5. RELATED WORK

As mentioned before, our framework refines Briand's framework [5] by taking into account AO specific composition models. A refinement of this framework was the subject of two other works. Zhao [18] implicitly extends the framework by formally describing new forms of dependencies between aspects and classes. Although it intends to be a generic framework for AO languages, it does not take into consideration a number of relevant AOP mechanisms, such as the ones supported by CaesarJ. Furthermore, it concentrates on coupling connections between aspects and classes, without mentioning coupling between the aspects themselves.

Bartsch and Harrison [4] explicitly extend Briand's framework targetting AspectJ. Their focus is on describing new types of connections and analysing the impact on other criteria. They have also created a new criterion specific for AspectJ, which was not available on Briand's framework.

We see our framework as a further development of these works for several reasons. First, our main goal, instead of targetting an specific language, is to create a framework which is generic enough to be language independent, but concrete enough to be easily instantiated for any AO language. Second, we realised that an extension of Briand's framework is not enough to cover new AO composition mechanisms, specially regarding CaesarJ's features. Hence, our approach was to analyse the new composition mechanisms and abstractions introduced by AO languages and, after a deep understanding of their properties and relationships, create structural and behavioural models, from which we derive our framework.

Third, we present a deep analysis of dynamic issues that affect the creation of coupling metrics, which was not present on other works. Fourth, we create a relevant categorisation for coupling connections. When new composition mechanisms are created, they can easily be grouped in our categories. Finally, our framework provides more expressiveness. For example, it is possible to define a coupling metric that measures the influence that exception handlers have on the coupling of a component. For such metric, one should select *Component* as *granularity* and *Exception Handler* as *aggregation.* This definition is not possible in other frameworks.

6. CONCLUSIONS AND FUTURE WORK

This paper has presented a unified framework for coupling measurements in AOP, where we have taken into account two sets of distinct AO composition models: the ones defined by AspectJ-like language and the ones that also support feature oriented decomposition mechanisms, such as CaesarJ. We have shown how the framework can be instantiated for Java, AspectJ and CaesarJ, and demonstrated the applicability of the framework by using it to define an existing coupling metric. We have also discussed the novelty of our work in comparison to related coupling framework propositions.

We intend to proceed our work in the furture in two distinct directions : *(i)* further development and *(ii)* empirical validation. The following items detail these possibilities.

Further Development

There are basically two directions to further develop the framework. First, we want to evalue to what extent the presented framework is generic enough to define in symmetric AO languages, such as Hyper/J. Second, we intend to assess how to include more dynamic coupling measurements in the framework, such as the ones defined in [3]. Up to now, the results seem to show that no major extensions and changes will be required in our framework.

Empirical Validation

Our framework has already been used to describe several existing coupling metrics. However, it lacks fundamental empirical validation. A first step is to validate that selecting framework criteria based on the external qualities that want to be measured will yield metrics that are valid, i.e., that correspond to this quality. More complex is the validation of the idea that it is possible to describe metrics with respect to an abstract model, instantiate it to several languages and generate results that are indeed comparable.

7. REFERENCES

[1] Multi-Language Coupling Framework for AOP, 2006. http://mulato.sourceforge.net.

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

[3] E. Arisholm, L. C. Briand, and A. Føyen. Dynamic Coupling Measurement for Object-Oriented Software. *IEEE Transactions on Software Engineering*, 30(8):491–506, 2004.

[4] M. Bartsch and R. Harrison. A Coupling Framework for AspectJ. In *Extended Abstract, To appear in Proc. Empirical Assessment of Sofware Engineering Conference - EASE.06*, 2006. *Extended Version Submitted for Publication* - http://www.sse.reading.ac.uk/people/M.Bartsch.htm.

[5] L. C. Briand, J. W. Daly, , and J. Wüst. A Unified Framework for Coupling Measurement in Object-Oriented Systems. *IEEE Transactions on Software Engineering*, 25(1):91–120, 1999.

[6] N. Cacho, C. Sant'Anna, E. Figueiredo, A. Garcia, T. Batista, and C. Lucena. Composing Design Patterns: A Scalability Study of Aspect-Oriented Programming. In *Proceedings AOSD'06*. ACM Press, 2006.

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

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

[9] A. Garcia, C. Sant'Anna, C. Chavez, V. Silva, C. Lucena, and A. von Staa. Separation of Concerns in Multi-Agent Systems: An Empirical Study. *Software Engineering for Multi-Agent Systems II, LNCS-2940, Springer-Verlag*, 2004.

[10] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, and A. von Staa. Modularizing Design Patterns with Aspects: A Quantitative Study. In *Proceedings AOSD'05*. ACM Press, 2005.

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

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

[13] 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*, pages 327–355. Springer, 2001.

[14] U. Kulesza, C. SantAnna, A. Garcia, R. Coelho, A. von Staa, and C. Lucena. Quantifying the Effects of Aspect-Oriented Programming: A Maintenance Study. In *Proceedings of 9th International Conference on Software Maintenance - ICSM06*, 2006.

[15] Li and Henry. Object-Oriented Metrics that Predict Maintainability. *Journal of Systems and Software*, 23:111 – 122, 1993.

[16] M. Mezini and K. Ostermann. Conquering aspects with Caesar. In *Proceedings AOSD'03*, pages 90–99. ACM Press, 2003.

[17] C. Sant'Anna, A. Garcia, C. Chavez, C. Lucena, and A. von Staa. On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework. In *Proceedings of Brazilian Symposium on Software Engineering (SBES'03)*, 2003.

[18] J. Zhao. Measuring Coupling in Aspect-Oriented Systems, 2004. *10th International Software Metrics Symposium (Metrics 04)*.