AVal: an Extensible Attribute-Oriented Programming Validator for Java

Carlos Noguera and Renaud Pawlak
INRIA - Futurs, Jacquard Project
LIFL, UMR CNRS 8022, Equipe GOAL - Bâtiment M3
59655 Villeneuve d'Ascq Cédex - FRANCE
{noguera, pawlak}@lifl.fr

Abstract

Attribute Oriented Programming (@OP) permits programmers to extend the semantics of a base program by annotating it with attributes that are related to a set of concerns. Examples of this are applications that rely on XDoclet (such as Hibernate) or, with the release of Java5's annotations, EJB3. The set of attributes that implements a concern defines a Domain Specific Language, and as such, imposes syntactic and semantic rules on the way attributes are included in the program or even on the program itself. We propose a framework for the definition and checking of these rules for @OP that uses Java5 annotations. We define an extensible set of meta-annotations to allow the validation of @OP programs, as well as the means to extend them using a compile-time model of the program's source code. We show the usefulness of the approach by presenting two examples of its use: an @OP extension for the Fractal component model called Fraclet, and the JSR 181 for web services definition.

1. Introduction

Attribute Oriented Programming[14] (@OP) allows the extension of a base language's semantics with concepts specific to a given domain by means of attributing the source code elements of the language with relevant metadata. For example, a class with an attribute `Persistent` may specify that the instances of that class must be saved in a persistent storage at runtime. Attribute-Oriented Programming can be implemented through library support (XDoclet, Apache Commons Attributes), or by direct language support (.Net, Java).

A set of attributes in an @OP framework can be seen as defining a Domain Specific Language (@DSL) on top of the base language. This @DSL imposes a set of syntactic and semantic rules in addition to those of the base language. The validations required to check these rules vary among @OP frameworks, and they can be quite complex. In the case of JSR-181[1], some annotations reference XML descriptors that must be validated by the framework; while for other annotations, validations must be performed on the code elements on which the annotations are placed; for example, a @WebService object cannot implement the `finalize()` method.

Although some support is included for the validation of these rules in existing frameworks and languages, in general it is not enough to cover the complex rules specified in current @OP languages. It is then up to the framework programmer to implement these supplementary checks in a way that is expressive, extensible, and provides meaningful error messages to the end user. In this paper, we present a framework for the validation of @OP programs which covers these properties.

1.1. Attribute Oriented Programming

Attribute-oriented programming is a program-level marking technique. Basically, developers can mark program elements (*e.g.*, classes, methods and fields) with attributes (*annotations*) to indicate that they maintain application-specific or domain-specific semantics [14]. Annotations separate the application's business logic from middleware-specific or domain-specific semantics (*e.g.*, logging and web service functions).

By hiding the implementation details of those semantics from program code, annotations increase the level of programming abstraction and reduce programming complexity, resulting in simpler and more readable programs. The program elements associated with annotations are transformed to more detailed programs by a supporting tool (*e.g.*, generation engine). For example, a generation engine may insert a call to a logging API into the methods associated with a `logging` annotation. The dependencies towards the underlying API are replaced by annotations, acting as weak

[1] Java Specification Request 181 for web service metadata

references. This means that any evolution of the underlying API is taken into account by the generation engine and the program code remains unchanged.

Recently attributes are used in a number of enterprise frameworks. Attributes are normally employed to embed in the source code information that was previously specified in external configuration files or derived from conventions of source code elements. For example, in the EJB3 [10] specification, annotations on JavaBeans coexist with the legacy XML-based descriptors. Also, in JUnit version 4, instead of relying on the naming convention that the name of test-case methods must start with the string `test`, an annotation @Test is used. Annotations in these two frameworks (i) enhance the readability of the source code, in EJB3, the programmer must only look in a single file to get all the information for the EJB; and (ii) make the use of frameworks easier, in JUnit[2], if the programmer misspells the @Test annotation, the Java compiler will flag the error, whereas this is not true for naming conventions. Annotations may also be used to directly represent DSLs. The aspect-oriented programming language AspectJ is a DSL closely based on Java, with the introduction of annotations in Java5, AspectJ migrated to being an annotation-based language by translating the DSL terms to equivalent annotations on pure Java code. By using AspectJ as a library rather than a language, the programmer can simplify her compilation process since both, aspects and base code, are processed by the same compiler.

The rest of the paper is organized as follows: in Section 2 we introduce Java 5 annotations as a platform for @OP and the Spoon annotation processor. In Section 3 we present our proposed framework for annotation validation, AVal; while in Section 4 we present two case studies: Fraclet (4.1) and an implementation of the JSR 181 (4.2). In Section 5 we evaluate the approach and in Section 6 compare it to related work. Finally conclude in Section 7.

2. Background

In this section we discuss Java 5 Annotations –with special attention to the validation capabilities provided within the language– and Spoon, our source code processor.

2.1. Java5 Annotations

In version 1.5, Sun included several language updates to Java, in particular, a metadata facility for program elements called *annotations*. Annotations in Java are a kind of typed metadata, that allows for @OP . They are defined in a way similar to interfaces, using the keyword `@interface`. Each annotation contains a number of **attributes** that can be primitive types (`int`, `float`, `String`), enumerations or classes represented as the return type of methods in the `@interface`. The semantics of the annotations can be implemented either at runtime (through the reflection API), or at compile-time (through the use of an annotation processor). An example of a Java5 annotation that allows programmers to specify which methods of a given class to test is shown in figure 1.

```
@Target({ElementType.CLASS,ElementType.METHOD})
public @interface Test{
  String name() default "";
}
```

Figure 1. Java annotation definition

Java5 annotations allow for a limited set of validations that restrict the source code elements on which the annotations can be placed. This is done by annotating the definition of the annotation (meta-annotating) with @Target. The Target annotation takes as argument an array with the elements on which the annotation can be placed. For example, the annotation defined in figure 1 can only be used on classes or methods. All other restriction on the usage of the annotations must be implemented by the @OP framework programmer. Depending on the way the @OP framework interprets the annotations, the misuse of an annotation will only be detected when the application is compiled or at runtime when the annotations are interpreted, if at all.

2.2. Spoon Annotation Processor

Spoon [11] is a source-code processor based on a metamodel of the program that models every code element, including statements and expressions. It relies heavily on generics to ensure type safe processing, and uses the concept of *processors* as units of program analysis and transformation. A processor is in essence an implementation of a visitor on the program's model. In each visiting, the processor has complete (both read and write) access to the model. Special processors are `AnnotationProcessors` that declare the annotation in which they are interested, and the type of elements on which the annotation is applied.

3. AVal Annotation Validator

To provide a generic and extensible framework for the checking of annotated programs in Java, we have implemented an *Annotation Validator* (AVal) as a Spoon processor. AVal follows the idea that annotations should describe the way in which they should be validated, and that self validation is expressed by *meta-annotations* (@Validators). This idea of meta-annotations as a way to describe the rules

[2] `http://www.junit.org`

of use of domain level annotations is a generalization of Java's @Target annotation discussed in section 2.1.

AVal's architecture is composed of four layers (figure 2):

Base program The (annotated) program that is to be validated. Elements of the program are annotated by attributes defined on the @DSL layer.

Domain Specific (Annotation) Language (@DSL) The domain specific annotations. Each annotation is meta-annotated by an AVal meta-annotation that expresses the rules for its validation.

@Validators Meta-annotations that encode the rules to validate annotations. Each @Validator represents a validation rule, and is itself annotated with the class that is responsible for the implementation of it.

Validation implementation A class implementing each validator. The class must implement the `Validator` interface, and it uses the Spoon compile-time model of base the program, @DSL annotation, and @Validator in order to perform the validation.

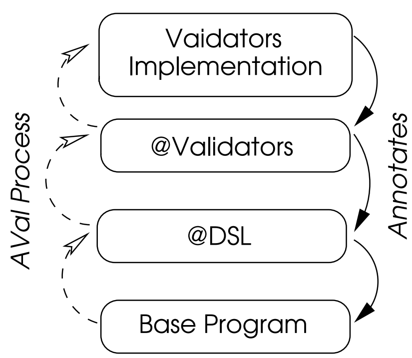

Figure 2. AVal Architecture

AVal is implemented as a Spoon source code preprocessor that is executed before the code generation or compilation phase in a @OP framework. It traverses the base code looking for domain specific annotations. Each time it finds an annotated element, it checks the model of the annotation's declaration to see if it has any @Validators. In case the annotation has one or more validators, the tool executes the each validator's implementation in the order in which they are defined. As a preliminary optimization, the validator's implementation is cached, so that if in the traversal of the program the same annotation is found twice, the correct validator implementation is executed without processing the annotation's definition again.

```
@Target({ElementType.METHOD})
@Inside(Documented.class)
public @interface WebLink{

  @URLValue  String value();

}

@Implementation(URLValueValidator.class)
public @interface URLValue{}
```

Figure 3. Example of the use of AVal @Validators: (@Inside and @URLValue)

3.1. A small example

In order to better explain the nature of AVal validations we show a small example. Take a small @DSL composed of the annotations @Document and @WebLink that allows to document classes by means of external web pages. As validation rules we define:

- Only methods can be annotated with @WebLink
- Only methods within *Documented* classes (classes annotated @Document) can be annotated with @WebLink.
- The @WebLink annotation contains an `String` attribute that must be a valid URL.

In order to automatically check these rules, we annotate (figure 3) the definition of the @WebLink with two @Validators: @Inside, that checks that all WebLinks are inside code elements annotated @Document, and @URLValue that states that the value of the annotation should be a valid URL. The definition of these two @Validators will be explained in detail in the following sections.

When the AVal processor is executed, it visits the program's elements checking if they have a @WebLink annotation. When it finds one, AVal opens the definition of the @WebLink annotation and processes the `value()` element; it then creates an instance of the `URLValueValidator` class and executes the implementation of the @Validator. A detailed description of this process is found in section 3.3.

3.2. Generic Validators

As a starting point, AVal defines a number of *generic* validators that can be used regardless of the domain. These generic validators are divided into structural and value validators.

3.2.1 Structural Validators

Structural validators reason on the relationship between annotations, and, between an annotation and the code element it annotates.

@AValTarget This validator extends Java5's @Target meta-annotation allowing for finer control on what elements can an annotation be used. With it, it is possible to state that an annotation can only be placed on interfaces, for example. The @Validator defines a single attribute:

- `value()`: The type of the target element. This type is expressed by the class that represents the element in Spoon's compile-time model. For example `CtInterface.class` if it is an interface.

@Inside This validator states that the current annotation must occur within the lexical scope of another annotation. For example, a method annotation @Foo can only be used on methods that belong to classes that are annotated @Bar, then @Foo is *inside* @Bar. This @Validator defines a single attribute:

- `value()`: The type of the parent annotation (For example `Parent.class`)

@Prohibits This validator states that if a code element is annotated with the current annotation, then it prohibits the use of another (given) annotation. This @Validator defines a single attribute:

- `value()`: The type of the forbidden annotation.

@Requires This validator is the dual of the @Prohibits. It states that the current annotation must occur in elements that are also annotated with another annotation. This @Validator defines a single attribute:

- `value()`: The type of the required annotation.

3.2.2 Value Validators

Value Validators reason on the attributes of the annotations. They allow to overcome Java5 restriction on the types allowed for annotation's attributes (primitive types, Classes, etc). Also they allow for domain-specific semantic checks.

@RefersTo This validator states that the value of the attribute must refer (be equal) to the value of another annotation somewhere in the program. This @Validator defines the following attributes:

- `value()`: The target annotation type to which the current attribute refers to.
- `attribute()`: The attribute in the target annotation type to which the current attribute refers to.

@Matches This validator applies on attributes of type `String`, and it checks that the values of the attribute match the provided Java regular expression.This @Validator defines a single attribute:

- `value()`: The regular expression.

@Unique This validator checks that a given attribute value is unique for a given program. That is, two program elements annotated with the same annotation type cannot have the same value in an Unique attribute. This @Validator defines no attributes of its own.

```
@A("http://localhost/")
public class Foo{
//...
}
```

a. Base Program

```
public @interface A{
 @URLValue
 String value();
}
```

b. @DSL annotation

```
@Implementation(URLValueValidator.class)
public @interface URLValue{}
```

c. @Validator

```
public class URLValueValidator
    implements Validator<URLValue>{

  public void check(
            ValidationPoint<URLValue> vp){

    String attribName = vp.getDslElementName();
    String value =
       (String)vp.getDslAnnotationValue(attribName);

    try{
       new URL(value);
    }catch(MalformedURLException ex) {
     //Report error
    }

  }
}
```

d. Validation implementation

Figure 4. Custom Validator

3.3. Extending Validators

We have found, as it will be discussed in section 4, that the generic validators shown before cover many validation needs. However, there are cases in which it is difficult or even impossible to translate a given domain rule into generic validators; for these cases we have implemented a way to extend the @Validator set for a particular domain.

New validators require two things: a new @Validator annotation, and its corresponding implementation. @Validators are normal Java annotations that are themselves annotated with their corresponding implementation. The implementation of a @Validator is a class that implements the `Validator` interface parametrized by the type of the @Validator. This interface defines a `check` method that is up called whenever the validated annotation is found. Validator implementations have access to the complete meta-model of the program, in particular to the annotated base program, the @DSL annotation and annotation definition, and the @Validator. These elements of the meta-model are encapsulated in a `ValidationPoint` object.

To illustrate the process of definition of a new validator, suppose that a domain rule requires that the value of the attribute `value()` of the annotation `A` is a valid URL (fig. 4b.). To this end, we define a @Validator URLValue (fig. 4c.). The class `URLValueValidator` is responsible for checking that the values of the @A are in fact URLs (fig. 4d.).

By providing access to the program's compile-time model and using Java, it is possible to implement complex validators, for example @RefersTo, that take into account the complete program, or validate with regard to external resources, such as checking an XML specification, or database schema.

4. Case Studies

In this section we present two Attribute-oriented specifications using Java5 annotations. Fraclet [12], an annotation framework for the Fractal component model [1] and the JSR181 for definition of web services [16]. We show how our proposal allows for a clear definition of the rules of use of these specifications.

4.1. Fraclet

Fraclet is an annotation framework for the Fractal component model. The Fractal component model defines the notions of *component*, *component interface*, and *binding* between components. Each of these main notions is reflected in the @DSL defined by Fraclet. There are two implementations of Fraclet, one using XDoclet2, and the other one using Java5 annotations and Spoon annotation processor. The annotations defined by Fraclet/Spoon are summarized in table 1.

The rules for the use of each of the annotations in Fraclet/Spoon are as follows:

@FractalComponent A Fractal component in Fraclet/Spoon is a Java **class** that defines a number of component attributes, bindings and operations. The @Target annotation provided by Java only allows to define that the annotation can be placed on types (classes or interfaces), therefore, the @Validator @AValTarget is used to restrict the Fractal components to being only classes. The complete definition of the annotation is shown below.

```
@AValTarget(CtClass.class)
public @interface FractalComponent {
  String controllerDesc() default "";
}
```

@FractalItf A Fractal business interface is a Java interface that defines a set of related operations in a component. The interface must contain a name that is unique for the application, and it must define if the interface is optional, and its cardinality. @Validators are provided to check all these rules:

```
@AValTarget(CtInterface.class)
public @interface FractalItf {
  @Unique  String name();
  Class signature() default None.class;
  @Matches("(singleton|collection)")
   String cardinality() default "singleton";
  @Matches("(mandatory|optional)")
   String contingency() default "mandatory";
}
```

@FractalAC A field annotated as FractalAC describes an attribute of the Fractal component, therefore, only fields that belong to a Fractal component class are allowed to be annotated @FractalAC. Also, since Fractal attributes and Fractal bindings are both represented using fields, it makes no sense to annotate a single field with both @FractalAC and @FractalBC. @Validators for these rules are included in the definition of the annotation:

```
@Inside(FractalComponent.class)
@Prohibits(FractalBC.class)
@Target(ElementType.FIELD)
public @interface FractalAC {
        String argument() default "";
        String value() default "";
}
```

@FractalBC A Fractal binding represents a binding between a component and a Fractal interface. The binding is represented as a field in a Fractal component class, and

Annotation	Location	Parameter	Description
@FractalComponent	Class	*controllerDesc*	Annotation to describe a Fractal component.
@FractalItf	Interface	*name, signature, cardinality, contingency*	Annotation to describe a Fractal business interface.
@FractalAC	Field	*argument, value*	Annotation to describe an attribute of a Fractal component.
@FractalBC	Field	*name, signature, cardinality, contingency*	Annotation to describe a binding of a Fractal component.
@FractalImportedInterface	Class	*interfaces*	Annotation to specify that the component provides a server interface which is not annotated with a @FractalItf.
@FractalRC	Field	-	Annotation to get the component part interface

Table 1. Overview of Fraclet annotations

therefore, is only valid in fields of classes annotated with @FractalComponent. It defines the name of the Fractal interface that is bound to (which must exist in the program), as well as the signature, cardinality, and contingency of the binding. These last three attributes follow the same rules than those of @FractalItf.

```
@Inside(FractalComponent.class)
@Prohibits(FractalAC.class)
@Target(ElementType.FIELD)
public @interface FractalBC {
  @RefersTo(value = FractalItf.class,attribute="name")
  String name();

  Class signature() default None.class;

  @Matches("(singleton|collection)")
  String cardinality() default "singleton";

  @Matches("(mandatory|optional)")
  String contingency() default "mandatory";

}
```

@FractalImportedInterface Fractal components implement interfaces that may not be Fractal business interfaces, but that still need to be exposed in the component; for example `java.lang.Runnable`. These interfaces are declared as *imported interfaces* in the definition of the Fractal component, therefore, it makes no sense to annotate a class with @FractalImportedInterface if it is not a Fractal component. Note that the `interfaces()` attribute is an array of @FractalItf, and therefore it is checked using the rules defined for Fractal business interfaces.

```
@Requires(FractalComponent.class)
public @interface FractalImportedInterface {
  FractalItf[] interfaces();
}
```

4.2. JSR 181

The JSR181 [16] is a specification for the description of web services using pure Java objects. The JSR defines a set of annotations and their mapping to the XML-Based Web Service Description Language. In section 2.5.1 of the specification, it is stated that implementations of the JSR must provide a validation mechanism that performs the semantic checks on the Java Bean web service definition. Table 2 summarizes the six of the ten annotations defined by the JSR.

Rules defined for the JSR describe restrictions not only on the use of the annotations, but also on certain properties of the annotated elements, for example that the web service implementation must not define a `finalize()` method, or that a *one-way* operation must have no return value. For this domain specific restrictions we extend the validation framework with a new @Validator for each annotation. This @Validator encapsulates all checks regarding the contents of the annotated element. The selected annotations of the @DSL are discussed below.

@WebService This annotation marks a Java class as a service implementation bean, or a Java interface as an endpoint interface. As the same annotation is used to describe two entities: service implementation and endpoint interface, the constraints on the annotated element vary depending on if the annotation is placed on a Java class or an interface. Regardless of where the annotation is placed, the `wsdlLocation()` attribute must be a valid URL.

If a class is annotated @WebService, it must be an outer class and it must not be `final` nor `abstract`, it must also define a default public constructor. These rules are validated by the `ValidWebServiceBean` @Validator. If an interface is annotated @WebService, it is required that the interface is `public` and the annotation is not allowed to define values for the `serviceName()` and `endPointInterface`. These rules are validated by the

Annotation	Location	Parameter	Description
@WebService	Class, Interface	*name, targetNamespace, serviceName, wsdlLocation, endpointInterface*	Class or Interface defining a web service
@WebMethod	Method	*operationName, action*	Method exposed as a web service operation
@OneWay	Method	–	Indicates that a given web server operation has only input messages and no output.
@WebParam	Method Parameter	*name, targetNamespace, mode, header*	Maps an individual operation parameter to a web service message
@WebResult	Method	*name, targetNamespace*	Maps the operation's return value to a web service result
@HandlerChain	Class, Interface	*file, name*	Associates an externally defined handler chain to a web service

Table 2. Overview of JSR-181 annotations

`ValidEndPointInterface` @Validator.

```
@Target( { ElementType.TYPE })
@ValidWebServiceBean
@ValidEndPointInterface
public @interface WebService {
  String name() default "";
  String targetNamespace() default "";
  String serviceName() default "";
  @URLValue
  String wsdlLocation() default "";
  String endpointInterface() default "";
};
```

@WebMethod This annotation marks a method as being a web operation for the web service. The method must be `public`, and its parameters and return type conform to the rules defined in the JAX-RPC specification [3]. The checks of the signature of the method are implemented in the `ValidWebOperation` @Validator.

```
@Target( { ElementType.METHOD })
@Inside(WebService.class)
@ValidWebOperation
public @interface WebMethod {
  String operationName() default "";
  String action() default "";
};
```

@Oneway This annotation states that a given WebMethod has only an input message, and no return value. The methods annotated @Oneway cannot declare checked exceptions, or define IN or INOUT parameters. The checks on the signature of the web methods are carried out by the `ValidOneway` @Validator.

```
@Requires(WebMethod.class)
@ValidOneway
public @interface Oneway {
};
```

@WebParam This annotation defines the properties for parameters of web methods. The specification does not define particular rules about this annotation other than that it must be defined only on parameters of web methods.

```
@Target( { ElementType.PARAMETER })
@Inside(WebMethod.class)
public @interface WebParam {
  public enum Mode {
        IN, OUT, INOUT
  };

  String name() default "";
  String targetNamespace() default "";
  Mode mode() default Mode.IN;
  boolean header() default false;
}
```

5. Evaluation

As discussed in previous sections, AVal has been applied to two @OP frameworks. In the case of Fraclet/Spoon we were able to check all the rules using only generic validators (3.2), without having to implement Fraclet-specific ones. Given that Fraclet/Spoon uses the same source-code transformation tool than AVal, we were able to integrate the @Validators seamlessly to the @OP framework. AVal is included in the latest release of Fraclet/Spoon.

In the case of the JSR181, rules about the relationship between annotations and rules that restrict the values of the annotation's attributes are encoded using generic validators, while restrictions on the program elements are validated by custom-made @Validators. It is interesting to note that the restrictions on the program elements are not directly related to the @DSL itself, but to the domain in which the annotations are used. Indeed, the rule that states that a class that implements a web service must not be final is independent of the way in which the class is marked as a web service. In this regard, the @Validators not only check annotations, but also domain related restrictions.

A weakness identified during the case studies is that, in order to attach the @Validators to the annotations defined in the @DSL, the developer needs to modify the definition of the annotations. This presents a problem when the @DSL is part of an external library (such as the Reference Implementation of the JSR181). This fact restricts the use of AVal to cases in which the @OP framework developer is also in charge of the @DSL definition, we do not believe this to be a too strong assumption.

5.1. Future Work

For the continuation of AVal, we expect to apply it to more complex @OP frameworks such as EJB3 (which defines more than fifty persistence annotations) and AspectJ5. The *AValidation* of these frameworks will allow us to verify and expand the number of generic validators, as well as to test the performance of the approach against large applications.

We would like also to explore the possibility of extending AVal to non-annotation based frameworks. The idea would be to annotate elements in the framework with @Validators that would check that they are correctly used. The tests would be similar as those performed by the @ValidEndPointInterface and @ValidWebServiceBean defined in Section 4.2.

6. Related Work

Static Validation Static validators allow developers to check properties of their code that go beyond of that what is provided by normal compilers. Lint [9] is one of the first tools to provide such checks by relying on (lightweight) static analysis. To reduce the amount of noise (false positives) that is normally generated by Lint-like tools, LCLint [5], and later Splint [6], guide the validation of programs through annotations (stylized code comments) that explicit programmer assumptions and intents. This use of annotations is comparable that of AVal's @Validators.

In [7], Hedin proposes an extensible, attribute-based static validator. In it, the grammar of a language is extended to check that custom programming conventions are followed. These extensions are similar in spirit to those possible with AVal; nevertheless, they lack the modularity and cohesion of implementing each extension in a separate class as is done in AVal, since the extension of the grammar is done by attributing each individual node of the AST and then acting upon these attributes, thus lacking locality.

By regarding validation as a crosscutting concern in a program's code, it is possible to encode it by means of Aspect Oriented techniques, this has been explored by Shomrat et. al. in [13]. Nevertheless, in an Aspect Oriented language such as AspectJ[8], no extra reflection facilities are provided, so the validation programmer must rely only on Java reflection which does not reify the body of methods and, since reflection is implemented at runtime, the @OP framework must be modified so that annotations are kept until runtime (using a special Java meta annotation). This restricts the domain of validations that can be performed

Annotation Validation Previous to the introduction of annotations in Java, XDoclet [15] relied on modified javadoc comments called tags to specify metadata for program elements. In XDoclet2, a form of tag validation is performed by tagging the tag definitions. The set of validations is fixed, and no special facilities are provided for extending them.

Cepa and Mezini's work [2] follows an approach similar to ours. They propose a meta annotation for the custom attribute facility in the .Net framework. However, they concentrate on *dependencies* between annotations (what we call structural validations 3.2.1) and do not foresee extensions to their model. In a later work [4], they propose an approach that is more general since it allows to validate constraints between different artifacts in the system (source code, configuration files, etc.). However,these constraints are expressed by means of a separate XML-based language, which in our opinion, goes against the idea of @OP which strives to reduce the use of external configuration files as much as possible.

7. Conclusion

We have presented AVal, an approach based on meta-annotations for the validation of the use of annotations in @OP Java applications. AVal is a declarative, expressive and extensible way to define and reuse validations of annotations. Also, it enhances the readability of @DSL source code definitions by embedding semantic information in their declaration. We have provided as case studies two @OP frameworks: Fraclet and the JSR181 for web services, and shown how to use AVal to include syntactic as well as semantic checks in them.

References

[1] E. Bruneton, T. Coupaye, and J.-B. Stefani. Recursive and Dynamic Software Composition with Sharing. In *Proceedings of 7th International Workshop on Component-Oriented Programming (WCOP'02)*, Malaga, Spain, June 2002.

[2] V. Cepa and M. Mezini. Declaring and enforcing dependencies between.NET custom attributes. In G. Karsai and E. Visser, editors, *GPCE*, volume 3286 of *Lecture Notes in Computer Science*, pages 283–297. Springer, 2004.

[3] R. Chinnici. *Java API for XML-based Remote Procedure Call (JAX-RPC) Specification*. Sun Microsystems, Oct. 2003. JSR-101.

[4] M. Eichberg, T. Schäfer, and M. Mezini. Using Annotations to Check Structural Properties of Classes. In M. Cerioli, editor, *Fundamental Approaches to Software Engineering, 8th International Conference*, volume 3442 of *Lecture Notes in Computer Science*, pages 237–252, Edinburgh, Scotland, 2005. Springer.

[5] D. Evans, J. Guttag, J. Horning, and Y. M. Tan. LCLint: A tool for using specifications to check code. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 87–96, 1994.

[6] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, Jan./Feb. 2002.

[7] G. Hedin. Attribute extensions - a technique for enforcing programming conventions. *Nord. J. Comput*, 4(1):93–122, 1997.

[8] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 26–35, New York, NY, USA, 2004. ACM Press.

[9] S. Johnson. Lint, a C Program Checker, 1978.

[10] L. D. Michel and M. Keith. *Enterprise JavaBeans, Version 3.0*. Sun Microsystems, May 2006. JSR-220.

[11] R. Pawlak. Spoon: annotation-driven program transformation — the AOP case. In *AOMD '05: Proceedings of the 1st workshop on Aspect oriented middleware development*, pages 1–6. ACM Press, 2005.

[12] R. Rouvoy, N. Pessemier, R. Pawlak, and P. Merle. Using attribute-oriented programming to leverage fractal-based developments. In *Proceedings of the 5th International ECOOP Workshop on Fractal Component Model (Fractal'06)*, Nantes, France, July 2006.

[13] M. Shomrat and A. Yehudai. Obvious or not?: regulating architectural decisions using aspect-oriented programming. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 3–9, New York, NY, USA, 2002. ACM Press.

[14] H. Wada and J. Suzuki. Modeling turnpike frontend system: A model-driven development framework leveraging UML metamodeling and attribute-oriented programming. In *MoDELS*, pages 584–600, 2005.

[15] C. Walls, N. Richards, and R. Oberg. *XDoclet in Action*. Manning Publications, 2004.

[16] B. Zotter. *Web Services Metadata for the Java Platform, Version 1.0*. BEA Systems, June 2005. JSR-181.