AVal: an Extensible Attribute-Oriented Programming Validator for Java

Carlos Noguera, Renaud Pawlak

AVal: an Extensible Attribute-Oriented Programming Validator for Java

Carlos Noguera
INRIA - Futurs, ADAM Project
noguera@lifl.fr
Renaud Pawlak
RPI, Hartford Campus
pawlar@rpi.edu

October 19, 2007

Abstract

Attribute Oriented Programming (@OP) permits programmers to extend the semantics of a base program by annotating it with attributes defined in an attribute domain-specific language (AttDSL). In this article, we propose AVal: a Java5 framework for the definition and checking of rules for @OP in Java. We define a set of meta-annotations to allow the validation of @OP programs, as well as the means to extend these meta-annotations by using a compile-time model of the program's source code. AVal is fully integrated into the Eclipse IDE. We show the usefulness of the approach by using examples of its use applied to three AttDSLs: an @OP framework that helps programming SAX parsers, an @OP extension for the Fractal component model called Fraclet, and the JSR 181 for web services definition.

1 Introduction

Attribute-oriented programming (@OP) is a program-level marking technique that allows developers to declaratively enhance the programs through the use of metadata. More precisely, developers can mark program elements (*e.g.*, classes, methods, fields) with attributes (*annotations*) to indicate that they maintain application-specific or domain-specific semantics [23]. In contrast to previous uses of attributes for program generation or transformation [13, 18], annotations are placed in the program *by the program developers*, as opposed to specialized tools, which use annotations to pass information from one processing phase to the next.

Annotations are usually represented as types that can contain a number of parameters, or *elements*, that serve as containers of the enclosed metadata. Annotations separate the application's business logic from middleware-specific or domain-specific semantics (*e.g.*, logging and web service functions). A set of annotations dedicated to

a given domain-specific concern can be referred to as an Attribute Domain-Specific Language (AttDSL).

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 toward the underlying API are replaced by annotations, which act as weak references and promote loosely coupled programs.

Because of the recent support of attributes (*annotations*) in C# and Java5, annotations are now being used in a number of enterprise frameworks. Annotations 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 [15] specification, annotations on JavaBeans coexist with the legacy XML-based descriptors. Also, in JUnit[1] 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 enhance the maintainability of the source code. In EJB3, the programmer must only look in a single file to get all the information for the EJB. In JUnit, if the programmer misspells the `Test` annotation, the Java compiler will flag the error, whereas this is not true for naming conventions.

Although @OP can help the final developers, defining a new AttDSL can be hard and requires discipline. In particular, the AttDSL specification should be made more explicit so that the @OP tools can verify that the AttDSL users do not misuse the annotations. Syntactic, typing, and semantic rules should be clearly defined to avoid hardly traceable compilation or runtime errors linked to unspecified AttDSL usage.

In this article, we propose AVal: a meta-annotation-based framework for defining and applying rules to AttDSLs. Through the study of a concrete AttDSL, we identify some common rules that shall be applied to this AttDSL, as well as extension mechanisms required to deal with specific cases. We provide a Java implementation that is fully integrated with the Eclipse IDE and apply it to our AttDSL as well as two other AttDSLs in order to prove the usefulness of our approach.

In Section 2, we first motivate our work by presenting Saxpoon: an @OP framework that defines a AttDSL and helps programming SAX parsers. We show how Saxpoon requires the definition of rules upon its annotations. In Section 3, we present AVal: a Java5 framework for the definition and checking of rules for @OP in Java that is used to implement the meta-annotations presented in Section 4, as well as the means to extend these meta-annotations by using a compile-time model of the Java program's source code. In Section 5, we show the usefulness of the approach by using examples of its use applied to two other AttDSLs: an @OP extension for the Fractal component model called Fraclet, and the JSR 181 for web services definition. Finally, we present some related work in Section 6 and discuss future work and conclusions in Section 7.

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

Annotation	**Location**	**Parameter**	**Description**
`XMLParser`	Class implementing `ContentHandler`	*dtd*	Marks a class as a Saxpoon class that handles XML files conforming with a dtd
`HandlesStartTag`	Method with arguments for each tag attribute	*tagName*	Method that handles the start of a tag
`HandlesEndTag`	Method	*tagName*	Method that handles the end of a tag

Table 1: Overview of Saxpoon annotations

2 Motivations

In this section, through the use of a simple AttDSL called Saxpoon, we present the main issues regarding @OP validation. Saxpoon defines three annotations that help the programmer to declaratively define SAX parsers. In spite of its usefulness, we see in this section that if the syntactic, typing, and semantic rules of Saxpoon are not explicited, the AttDSL can be easily misused by programmers. These misuses will lead to errors that are difficult to trace, because they are not identified by the AttDSL as breaking a well-defined rule. Hence, we will come to the idea that motivates our work and will lead us to specifying well-identified rules for AttDSLs.

In Section 2.1, we introduce Saxpoon and explain how it helps with the programming of SAX-based XML parsers in Java. By analyzing the Saxpoon example, Section 2.2, formulates our main problem statements that call for clarifying the AttDSLs semantics in order to ensure that programmers use annotations correctly.

2.1 AttDSL Example: Saxpoon

To better explain the nature of @OP and to illustrate and motivate our work, we present a simple @OP framework, called Saxpoon, that we have developed as a test-bed for annotation validation. Saxpoon is a compile-time @OP framework that aids the programmer in the construction of XML manipulation classes that use the *Simple API for XML (SAX)* [14] in Java. SAX is an *event-oriented* API that defines, in a `ContentHandler` interface, a number of call-back methods that the programmer must specialize in order to extract information from an XML file. Among the events emitted by a SAX parser, `startElement()`, `endElement()` and `characters()` deal with the opening, closing and the text in between tags. This means that if the programmer is interested in the start of several tags, she must place the tag handling code for each tag on the same `startElement()` method, which reduces its cohesion due to, normally, a large `if` that distinguishes between the different tags.

SAX can validate that an XML document conforms to a DTD, but it cannot ensure that a SAX `ContentHandler` Java implementation actually supports a given DTD. Saxpoon tries to resolve these problems by using annotations to declaratively tune the use

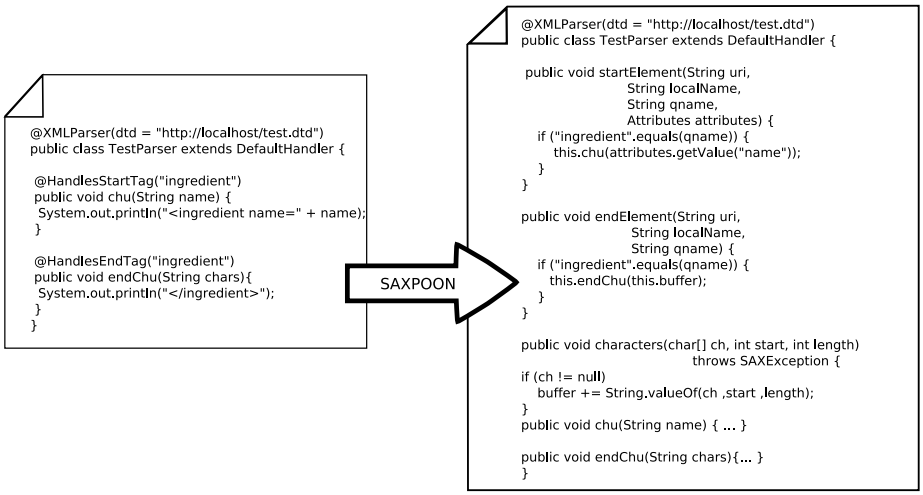

Figure 1: Saxpoon Transformation

of SAX. Compile-time validation and code generation will reduce the risk of implementation faults and the need of programming repetitive infrastructural code.

Table 1 shows the three annotations defined by Saxpoon. `XMLParser` marks a class implementing the `ContentHandler` interface as a Saxpoon class. It takes as a parameter the DTD file that describes what the class can handle, which can be used to automatically generate code to validate XML with the DTD. Also, at compile-time, Saxpoon must validate that the content handler actually provides support to parse the given DTD. This is possible because Saxpoon uses the `HandlesStartTag` and `HandlesEndTag` annotations to declare the handled tags. Methods annotated with `HandlesStartTag` belong to a class annotated by `XMLParser` and handle the start of a given tag. Methods annotated with `HandlesEndTag` are supposed to handle the event generated when the tag they handle is closed. Using these annotations, Saxpoon will generate the consolidated `start`, `end`, and `character` methods upon compilation, as shown in Figure 1. More than contribute to the state of the art of XML technology, Saxpoon's goal is to provide a concrete example of a simple, yet not trivial annotation DSL.

2.2 Problem Statement

Although Saxpoon AttDSL is only composed of the three tags summarized in Table 1, it implies a number of rules of use. Here is the list of rules that a Saxpoon program should follow:

1. Only classes can be marked as `XMLParser` types.
2. Methods shall never be marked with `HandlesStartTag` and `HandlesEndTag` at the same time.

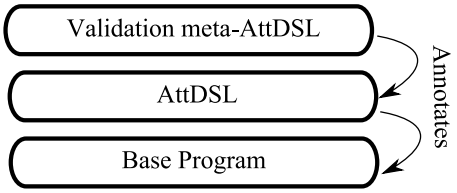

Figure 2: Relationship between base program, AttDSL and meta-AttDSL

3. Methods marked with `HandlesStartTag` or `HandlesEndTag` will only be translated if they belong to a class marked with `XMLParser`.
4. Methods marked with `HandlesStartTag` or `HandlesEndTag` must be of type **`void`**, their parameters must be of type `String`, and they cannot throw any checked exceptions.
5. `HandlesStartTag` and `HandlesEndTag` annotations must take as argument a tag name defined in the DTD passed as argument to the containing `XMLParser` annotation.
6. The parameter names on methods marked with `HandlesStartTag` must match with attributes of the handled tag as defined in the DTD.
7. Methods marked `HandlesEndTag` must take a single parameter that will stand for the characters within the marked type.

From the perspective of the Saxpoon user, these restrictions are not made explicit since they are not present in the annotation type definitions, and it is up to her to refer to the relevant documentation. From the perspective of the programmer that wrote the Saxpoon transformation engine, the checks for these restrictions are tangled with the source code transformations themselves, making their thorough checking a matter of programming discipline.

In this article, we claim that it is possible to provide an explicit way to describe the annotations' semantics, that will then be able to be verified by a generic tool. This will allow the AttDSL programmers:

1. to explicitly and more completely describe the annotations,
2. to reuse a generic tool for all the AttDSL instead of having to program a new ad-hoc static analysis engine for each AttDSL, and
3. to separate the rules *validation* phase from the interpretation phase of the AttDSL execution.

3 AVal annotation validation Framework

To perform @OP validation we decided to apply the concept of @OP itself by defining an AttDSL that contains a set of meta-annotations for the domain of *annotation*

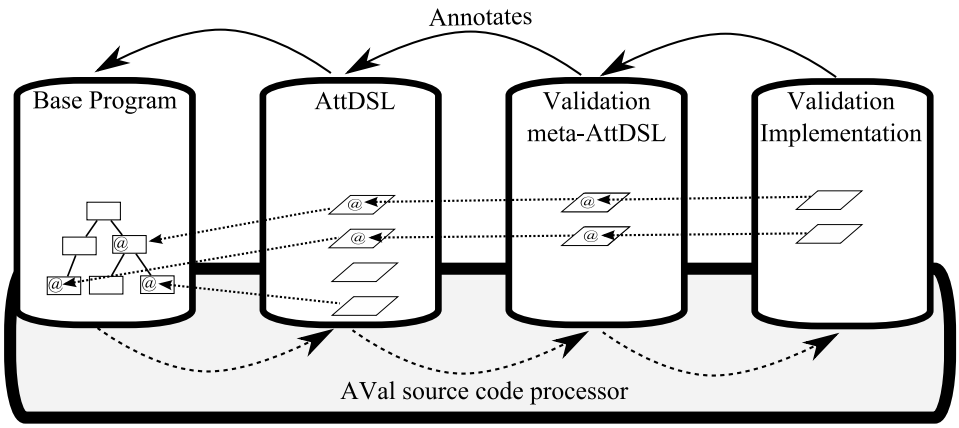

Figure 3: AVal Architecture

rules validation. These validation meta-annotations are used to mark the definition of the subject AttDSL with meta-data relevant to validating a given rule, as can be seen in Figure 2. For example, if we consider the definition of Saxpoon presented in Section 2, we have identified the rule that methods marked with `HandlesStartTag` and `HandlesEndTag` will only be translated if they belong to a class marked with `XMLParser`. A generic way to express this kind of rules is to use a meta-annotation `Inside`, that we apply to `HandlesStartTag` and `HandlesEndTag` to specify that these annotations should only be found *inside* `XMLParser`.

Annotating annotations with other annotations has the advantage to make the rules *explicit* in the AttDSL definition and *local* to the annotation they apply to. Furthermore, it recursively applies the principles of @OP to itself, thus forming a *meta-circular* system, which is a consistency indicator. The advantages of @OP, such as declarative programing and loose coupling can then be directly applied to our approach.

AVal is a Java implementation of a validation meta-AttDSL. It provides a number of built-in validations, and means to add custom ones. It uses Spoon [16] for compile-time reflection and annotation processing, and through it, provides integration to the Eclipse IDE.

3.1 Annotation Validation for Java

The concept of using meta-annotations to declare the restrictions of use of Java annotations is already included in the JDK. Indeed, the Java Language Specification [8] defines a `Target` annotation that must be placed on annotation type definitions to restrict where instances of the annotation type can be placed. However, asides from `Target`, no other validation annotations are provided.

3.2 Architecture

AVal's architecture is composed of four layers (Figure 3):

Base program: The (annotated) program that is to be validated. Elements of the program are annotated by annotations defined on the AttDSL layer.

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

Validation meta-annotations: AVal annotations that encode the rules to validate domain specific annotations. Each meta-annotation represents a validation rule, and is itself annotated with the class that is responsible for the implementation of it.

Implementation: A class per validation meta-annotation. The class must implement the `Validator` interface, and it uses the Spoon compile-time model of base the program, AttDSL annotation, and meta-annotation in order to perform the validation.

AVal is implemented as a Spoon source code pre-processor that is executed before the code generation or compilation phase in an @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 meta-annotations. In case the annotation has one or more validators, the tool executes each implementation in the order in which they are defined. As a preliminary optimization, the implementation is cached, so that if in the traversal of the program the same annotation is found twice, the correct implementation is executed without processing the annotation's definition again.

3.3 Problem Fixers and Error messages

Built-in meta-annotations in AVal contain a number of special parameters, which will be defined on Section 4. Aside from these, all the meta-annotations included in AVal state three convenience elements: `message`, `severity`, and `fixers`. These elements permit the AVal user to adjust the presentation of the errors to a particular AttDSL. Each of these convenience elements is explained below.

Error Messages AVal allows the programmer to customize the messages raised by failed validations in two ways: first, the severity of the message can be presented either as an `ERROR`, a `WARNING` or a `MESSAGE`. Second, the text of the message can be customized to better fit the context of the @OP subject of validation, to this end, a simple template language is defined. Both these customizations are realized when the AVal meta-annotation is used on an annotation type definition by providing values to the `severity` and `message` elements. For example, in Saxpoon, the definition of the `HandlesStartTag` is annotated with a `RefersTo` meta-annotation to raise a warning when the start of a given tag is handled but not the end event:

```
public @interface HandlesStartTag {
  @RefersTo(value=HandlesEndTag.class,
            message="No handler defined for the end of <?val> tag",
```

```
            severity=Severity.WARNING)
  String value();
}
```

Problem Fixers With Spoon (the annotation processor used by AVal), whenever an error is reported to the environment, it is possible to provide a set of source code transformations that can fix the error. These transformations, or *problem fixers* as defined in the Spoon API, are classes implementing the `ProblemFixer` interface. They are applied interactively by the user through the IDE (in our case Eclipse), and when invoked, a problem fixer can manipulate the program's AST by using the Spoon API .

For example, consider the `HandlesStartTag` annotation in Saxpoon. This annotation, by means of the `RefersTo` meta-annotation, will produce a warning whenever no corresponding method to handle the closing of its tag is found. In this case, a way to aid the programmer would be to produce a stub of the missing method. This can be implemented via a problem fixer, which is affected to the annotation as follows:

```
public @interface HandlesStartTag {
  @RefersTo(value=HandlesEndTag.class,
            message="No handler defined for the end of <?val> tag",
            severity=Severity.WARNING
            fixers={AddEndHandlerStub.class})
  String value();
}
```

Problem fixers allow the programmer of the base application to choose a *predefined* source code snippet template that help him to fix an error. The transformation is then applied on the base program so that the programmer can customize the snippet. In the case of the `HandlesStartTag` with out its corresponding end handler, a method (with the correct signature and annotation) is added. It is up to the programmer to write the code to handle the end of the tag. The problem fixers are interactively invoked through the IDE by the programmer.

3.4 Extending Validations

Even though the validations defined so far cover many of the validation needs, there are cases in which it is difficult or even impossible to translate a given domain rule into generic validators; for these cases it is possible to extend the meta-annotation set for a particular domain.

New validators require two things: a new AVal meta-annotation, and its corresponding implementation. Meta-annotations are normal Java annotations that are themselves annotated with their corresponding implementation. The implementation of a meta-annotation is a class that implements the `Validator` interface parametrized by the type of the meta-annotation. 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 AttDSL annotation and annotation definition, and the meta-annotation. These elements of the meta-model are encapsulated in a `ValidationPoint` object. For example, a new

validation annotation, and corresponding implementation, for checking that a value is a valid URL would take this form:

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

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

  public void check(
            ValidationPoint<URLValue> vp){
     // validation and error reporting...
  }
}
```

As a more complex example, consider a validation that ensures that the method on which the annotation is placed does not throw any unchecked (*i.e.*, runtime) exceptions. This example takes advantage of the Spoon API that allows the programmers to introspect the code inside the body of a method, thus allowing more complete static analysis than with `apt` for instance (see Section 6 for further discussion).

```
@Implementation(NoUncheckedExceptionsValidator.class)
public @interface NoUncheckedExceptions {}

public class NoUncheckedExceptionsValidator
             implements Validator<NoUncheckedExceptions>{
    public void check(ValidationPoint<CheckNoUncheckedExceptions> vp) {
        // get the method on which the annotation is placed
        CtMethod<?> meth = (CtMethod<?>)vp.getProgramElement();
        // get all the throw clauses that throw an unchecked exception
        List<CtThrow> matches = Query.getElements(meth.getBody(),
            new UncheckedExceptionsFilter(outParam.getReference()));
        if(!matches.isEmpty()) {
            //report a warning on each throw clause
        }
    }
}
```

In the previous code, the `check` method uses the Spoon API to run a filter-based query on the body of the annotated method. A query scans the AST to return the nodes that match the given filter. Here `UncheckedExceptionsFilter` will match any occurrence of a `CtThrow` node which thrown expression is a subtype of `RuntimeException`. In addition to this, the filter implementation can check that the thrown exception is not caught within the method's body. Although this analysis is still local to the method body, it would also be possible to implement an inter-procedural control-flow analysis. However, the point here is not to discuss complex static analysis, but more to show that the full program AST is required when coming to implement more complex validations on the program.

3.5 Library annotations

So far, in order to use AVal on a given AttDSL, the source code of the annotation types is needed. Indeed, since the validation relies on meta-annotations, the AVal programmer must be able to add and remove annotations to the AttDSL. This, in principle,

is a very strong limitation, since most of the AttDSL users have no control over the definition of the AttDSLs, see EJB3 annotations for example. To overcome this issue, in AVal it is possible to add validations to annotations for which the source code is not present by *replacing* those annotations during the validation phase. The idea is to rewrite the annotation type definition, and use a `ReplaceAnnotationInPackage` annotation to temporarily change the package of the new annotation. After the validation round is over, replaced annotations are deleted from the model, restoring their original implementation.

To illustrate this, consider the `java.lang.SuppressWarnings` annotation. It is defined in the Java API to instruct the compiler to suppress certain warnings produced inside annotated elements; however, the documentation of this annotation warns: "programmers should always use this annotation on the most deeply nested element where it is effective. If you want to suppress a warning in a particular method, you should annotate that method rather than its class.". Indeed, spurious use of this annotation (for example, placing it on a package) may make the compiler disregard important, unintended warnings. A way to avoid this case could be to restrict the `SuppressWarnings` to a finer grain.

AVal can be used to further restrict the `SuppressWarnings` to methods only by including the following annotation type definition on a `dummy` package and replacing the one in `java.lang`:

```
package dummy;

@ReplacesAnnotationInPackage("java.lang")
@AValTarget(CtMethod.class)
public @interface SuppressWarnings{
        String[] value();
}
```

By doing this, whenever the AVal processor finds a `java.lang.SuppressWarnings` annotation, it will perform the checks required by the `dummy.SuppressWarnings` as directed by the `ReplaceAnnotationInPackage` annotation; namely check that the annotation is placed on a method.

3.6 Eclipse Integration

AVal is integrated with the Eclipse IDE through the Spoon JDT plug-in[2]. This plugin enables Spoon processors to be applied on a given Eclipse project each time it is compiled. By doing this, the relevant @OP validations are applied seamlessly as dictated by the meta-annotations present on AttDSLs that the programmer uses. Error and warning messages are displayed in the same way as those raised by the Java compiler, and problem fixers are displayed as Eclipse's quick fixes. This integration, for a Saxpoon program, is shown in Figure 4.

[2]http://spoon.gforge.inria.fr/Spoon/Installation

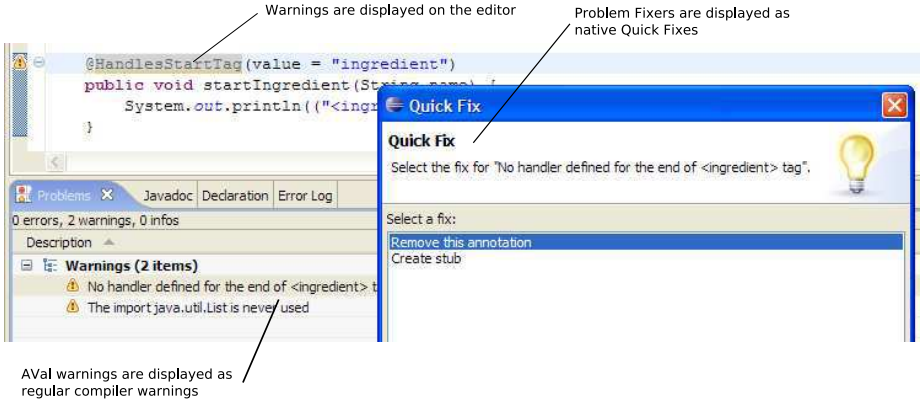

Figure 4: AVal integration with Eclipse IDE

4 Meta-Annotation-based Validation

By studying Saxpoon and other AttDSLs, we have defined a generic meta-AttDSL composed of seven meta-annotations: `Inside`, `Prohibits`, `Requires`, `RefersTo`, `Matches`, `AValTarget` and `Type`. These define a meta-AttDSL for annotation validation, which covers most of the basic needs for typical annotation validation. We have implemented this language in AVal, but these particular validations could be implemented by other annotation processors such as the ones discussed Section 6.

Meta-annotations are separated in two groups: the ones that express constraints between annotations in the subject AttDSL and the ones that express constraints between the AttDSL's annotations and the program on which they are placed. It is important to note that this separation is only for illustrative purposes, and that all validations are of the same *kind*.

To better explain the meaning and use of the proposed meta-annotations, the next sections also give the formal semantics for each of them.

4.1 Notations and Definitions

Let n be a node in the AST of a program and $ntype$ a function that maps n to the kind of element that it represents ($Class_N$, $Interface_N$, $Enum_N$, $Method_N$, $Field_N$, $Annotation_N$, $AnnotationElement_N$). Node types are partially ordered by the subtype relation $<:_N$ and nodes of the AST are partially ordered by the transitive ancestor relation $<_T$. Let $annot$ be a function that maps nodes of the AST to the set of annotation instances in that node. Let a an annotation and $type$ a function that maps a to the actual type of the annotation. Types are partially ordered by the transitive subtype relation $<:$. An array of type t is noted $t[]$. Finally, the function def maps annotation instances, or their elements, to the AST node in which they are defined.

Annotation instances in nodes and their definition must be defined (*def*) by an

annotation node, which is part of the AST:

$$\frac{type(a) <: Annotation}{ntype(def(a)) = Annotation_N}$$

Annotations can define annotation elements and the arguments of the annotation instances (dotted notation) have their corresponding definition nodes in the AST also:

$$\frac{type(a) <: Annotation}{ntype(def(a.e)) = AnnotationElement_N}$$

4.2 Validation with regards to annotations

These meta-annotations define restrictions on where the AttDSL annotations can be placed with respect to other annotations: `Inside`, `Prohibits`, `Requires`, or restrictions on the values of their elements: `RefersTo`, `Matches`.

`Inside` When an annotation instance a is of a type annotated with an `Inside` meta-annotation in that refers to another annotation type B, the use of the annotation a on an AST node n is valid only if it occurs on an AST node that has a (indirect) parent node annotated by an instance of B. The `Inside` annotation defines a single element `value` that contains the containing annotation type.

$$\frac{\begin{array}{c} type(a) <: Annotation \\ type(in) = Inside \qquad in \in annot(def(a)) \qquad in.value = B \qquad n \in AST \end{array}}{a \in annot(n) \rightarrow \exists m \in AST\ (m <_T n \wedge (\exists b : B\ (b \in annot(m))))}$$

A typical application of this meta-annotation is given by Saxpoon to implement the rule that methods marked with `HandlesStartTag` and `HandlesEndTag` will only be translated if they belong to a class marked with `XMLParser` (rule 3 of Section 2.2). So, in Saxpoon, `HandlesStartTag` and `HandlesEndTag` is meta-annotated with `Inside(value=@XMLParser)`.

`Prohibits` Given a node n that is annotated by an instance a whose annotation type is itself annotated by an instance pr of type `Prohibits` with an argument B prevents instances of B to annotate n. The `Prohibits` annotation defines a single `value` element that contains the prohibited annotation type.

$$\frac{\begin{array}{c} type(a) <: Annotation \\ pr : Prohibits \in annot(def(a)) \qquad pr.value = B \qquad n \in AST \end{array}}{a \in annot(n) \rightarrow \neg\ \exists b : B\ (b \in annot(n))}$$

Saxpoon's rule 2 of Section 2.2 gives us a good application of this meta-annotation. This rule states that no methods can be marked with `HandlesStartTag` and `HandlesEndTag` at the same time. So, in Saxpoon, `HandlesStartTag` is meta-annotated with `Prohibits(value=@HandlesEndTag)` and `HandlesEndTag` is meta-annotated with `Prohibits(value=@HandlesStartTag)`.

`Requires` This annotation is the dual of `Prohibits`. It requires that all nodes n, annotated with an annotation instance a whose type has an annotation re of type `Requires`, to be also annotated with an instance of its argument B. The `Requires` annotation defines a single `value` element that contains the required annotation type.

$$\frac{\begin{array}{c} type(a) <: Annotation \\ re : Requires \in annot(def(a)) \qquad re.value = B \qquad n \in AST \end{array}}{a \in annot(n) \rightarrow \exists b : B \ (b \in annot(n))}$$

`RefersTo` An instance of this annotation is placed on an annotation element $def(a.i)$ of an annotation type of the AttDSL. It states that the values of the annotated element on an annotation instance a must be equal to the value of an annotation instance of type B present in the AST. The `RefersTo` contains two elements: `type` that defines referred annotation type, and `id` that defines the argument to which $a.i$ must point to, which defaults to `value`.

$$\frac{\begin{array}{c} type(a) <: Annotation \\ rt : RefersTo \in annot(def(a.i)) \qquad rt.type = B \qquad rt.id = j \end{array}}{\forall n_1 \in AST \ (a \in annot(n_1) \rightarrow \exists n_2 \in AST \ (n_1 \neq n_2 \wedge \exists b : B \in annot(n) \ (a.i = b.j)))}$$

This annotation is not used in Saxpoon. It is, however, used on the Fraclet component AttDSL presented Section 5.3 to specify bindings between components. Indeed, each annotation that defines a binding must state to which component it will be bound to. This is verified by annotating the binding annotation with `RefersTo(Component,name)`.

`Matches` Instances of this annotation, also placed on an annotation element $def(a.e)$ on an annotation type of the AttDSL, restrict the values of this element $a.e$ on annotation instances a to those that match the regular expression that is defined on the value $ma.value$ of the `Matches` instance ma. `Matches` contains a single element `value` which stands for the regular expression string.

$$\frac{\begin{array}{c} type(a) <: Annotation \\ ma : Matches \in annot(def(a.e)) \qquad ma.value = x \qquad n \in AST \end{array}}{a \in annot(n) \rightarrow \ matches(a.e, x)}$$

Saxpoon's AttDSL shown in Table 1 give us a context where this meta-annotation can be useful. Indeed, `XMLParser` defines an element `dtd` which should contain an URL string. Thus, we can meta-annotate `dtd` with `Matches`, where the value would be a regular expression that matches well-formed URL strings.

4.3 Validations with regards to source code

These meta-annotations express restrictions on the locations in the program in which AttDSL annotations can be placed, with respect to the program elements themselves.

AValTarget This annotation restricts the type T_N of nodes of the AST on which an annotation a of a given annotation type can be placed. This meta-annotation defines a single `value` element which contains the node type.

$$\frac{type(a) <: Annotation \quad at : AValTarget \in annot(a) \quad at.value = T_N \quad n \in AST}{a \in annot(n) \rightarrow\ ntype(n) = T_N}$$

Type This annotation restricts the (program) type T on which a certain annotation a can be placed. Depending of the type of AST node n, $type(n)$ denote different elements: if the node n is a Type (i.e. `CtClass`, `CtInterface`, etc) then the $type$ function is the type that the class or interface represents. If the node n represents a method or a constructor (`CtExecutable`) then the $type$ function evaluates to the return type of the method[3] or constructor, and if the node n is a field (`CtField`), then the $type$ function is the type of the field. `Type` defines a single element `value` which contains the program type.

$$\frac{type(a) <: Annotation \quad t : Type \in annot(a) \quad t.value = T \quad n \in AST}{a \in annot(n) \rightarrow\ type(n) = T}$$

Saxpoon's rule 4 of Section 2.2 can be partially implemented by this meta-annotation, by marking `HandlesStartTag` and `HandlesEndTag` with `Type(value=Void)`.

4.4 Implementing validations

To illustrate how these validations can be implemented, we next show the AVal validation classes for the `Prohibits` and `Type` meta-annotations. The implementation for the other annotations can be viewed at AVal's website[4].

```
1 public class ProhibitsValidator implements Validator<Prohibits> {

3 public void check(ValidationPoint<Prohibits> vp) {
4  Class<? extends Annotation> prohA = vp.getValAnnotation().value();
5  CtElement element = vp.getProgramElement();
6  boolean valid = element.getAnnotation(prohA) ==null;
7   if (!valid) {
8     //report error
9   }
10  }
11 }
```

The `check()` method is called each time an base-program annotation that has the `Prohibits` meta-annotation is found. In it, the forbidden annotation is obtained in line 4, and the element being analyzed in line 5. A check is then performed to see whether the element carries the forbidden annotation or not. If it does, an error is reported

[3] for methods returning **void**, a $Void$ type is used

[4] http://spoon.gforge.inria.fr/AVal/Main

```
1 public class TypeValidator implements Validator<Type> {

3   public void check(ValidationPoint<Type> vp) {
4     Class<?> actualClass = null;
5     if (vp.getProgramElement() instanceof CtType<?>) {
6         actualClass = ((CtType) vp.getProgramElement()).getActualClass();
7     }
8     if (vp.getProgramElement() instanceof CtTypedElement<?>) {
9         CtTypedElement<?> typed = (CtTypedElement<?>) vp
10                                        .getProgramElement();
11        actualClass = typed.getType().getActualClass();
12    }
13    if (!vp.getValAnnotation().value().isAssignableFrom(actualClass)) {
14      //report error
15    }
16  }
17 }
```

In this case, two separate cases are taken into account, as specified in the annotation's definition in the previous section: either base-program element is a type, (line `5`), or it is a typed expression (line `8`). Once the element's type is established, it is check whether it is type-compatible with `Type`'s parameter.

5 Applications

In the previous section, we have defined seven meta-annotations, which are generic enough to be applied to many AttDSLs in order to explicit their rules in a declarative and modular way. In this section we present the application of our AVal Java implementation to three attribute-oriented frameworks: Saxpoon, our running example; the set of annotations for the definition of web services included in Sun's JSR 181, and Fraclet, an @OP framework for the Fractal component model. Through these three case studies we demonstrate the usefulness of the approach.

5.1 Saxpoon

As presented in Sections 2.1 and 2.2, the Saxpoon AttDSL defines a number of restrictions on the use of its annotations. In order to validate that these restrictions are met, we apply for each Saxpoon annotation, AVal meta-annotations accordingly. Custom annotations are written to deal with validations regarding DTD documents and source code conventions, while generic validations, (*e.g.*, those described in Section 4), check restrictions on the structure of the annotations. The custom meta-annotations defined are: `ValidDTD`, `ValidStartTag` and `ValidEndTag`, they rely on the DTD defined by the `XMLParser` annotation. The use of AVal to validate in Saxpoon annotations is explained below.

XMLParser This annotation marks a class implementing the `ContentHandler` interface as a Saxpoon class, which is validated by `Type`. It takes as a parameter the DTD file that describes what the class can handle. The `dtd()` annotation element must be an URL (`URLValue`) pointing to a valid DTD file. The DTD validation is handled by `ValidDTD`.

```
@Type(ContentHandler.class)
public @interface XMLParser {
        @URLValue
        @ValidDTD
        String dtd() default "";
}
```

HandlesStartTag States that a method on a class annotated by `XMLParser` handles the start of a given tag, validated by `Inside`. The methods annotated by this annotation must be **void** and define only `String` arguments whose names match attributes of the handled tag as defined in the DTD. These constraints on the signature of the methods annotated with `HandlesStartTag` are checked by `ValidStartTag`. Finally, as a help to the developer, a warning is raised when the event of start of a tag is handled, but not its corresponding end event (`RefersTo`).

```
@Inside(XMLParser.class) @ValidStartTag
public @interface HandlesStartTag {
        @RefersTo(value=HandlesEndTag.class,
                        message="No handler defined for the end of <?val> tag",
                        severity=Severity.WARNING)
        String value();
}
```

HandlesEndTag Methods that contain this annotation are supposed to handle the event generated when the tag they handle is closed. They must be of return type **void** and take a single `String` argument which stands for the characters encountered between the start and end of the handled tag, checked by `ValidEndTag`. A warning is also raised if the corresponding start tag is not handled (`RefersTo`).

```
@Inside(XMLParser.class) @ValidEndTag
public @interface HandlesEndTag {
        @RefersTo(value=HandlesStartTag.class,
                        message="No handler defined for the start of <?val> tag",
                        severity=Severity.WARNING)
        String value();
}
```

It is interesting to further discuss the role of the `Inside` meta-annotation Saxpoon AttDSL. Given that the validations on the methods that handle start/end tags depend on a particular DTD which is defined in the `XMLParser` annotation, it makes no sense for a handler to be *outside* of the lexical scope of an `XMLParser` annotation. The rule, however, does not state inside of which `XMLParser` annotation, nor how deep inside. Indeed a class annotated `XMLParser` could have an inner class whose methods are annotated

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

`HandlesStart/EndTag`. Whether or not this is a semantic error depends on the Saxpoon programmer's intentions. In case it is, another meta-annotation that checks that the handling method is actually a *member* of the XML parsing class would be needed.

By combining AVal-defined validations with domain-specific validations, restrictions on Saxpoon's annotations are explicitly stated, and automatically checked each time the code is generated. As a side effect, since the rules of use of the annotations is encoded in their annotations, the annotations become self-documented.

5.2 JSR 181

The JSR 181 [25] 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. We show how these validations can be implemented using AVal. Table 2 summarizes 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 these domain-specific restrictions we extend the validation framework with a new meta-annotation for each annotation. This meta-annotation encapsulates all checks regarding the contents of the annotated element. The selected annotations of the AttDSL 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()` element 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` meta-annotation. 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 `ValidEndPointInterface` meta-annotation.

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

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

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 OUT or INOUT parameters. The checks on the signature of the web methods are carried out by the `ValidOneway` meta-annotation.

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

```
@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;
}
```

`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.

With the use of AVal on JRS181's annotations, we show that the approach is scalable to industry-defined annotation libraries. This application of AVal also shows that Aval can be used on externally defined AttDSLs by using the extension mechanism explained in Section 3.4. From these particular meta-annotations new generic meta-annotation could be derived in the future. To illustrate how this can be achieved, consider the `ValidOneWay` annotation present on the `OneWay` JSR 181 annotation. The validation is implemented by the `ValidOnewayValidator` class:

```
public class ValidOnewayValidator implements Validator<ValidOneway> {

  public void check(ValidationPoint<ValidOneway> vp) {
    if (vp.getProgramElement() instanceof CtMethod) {
      CtMethod<?> oneway = (CtMethod<?>) vp.getProgramElement();

      boolean isVoid = oneway.getType().equals(
        oneway.getFactory().Type().createReference(void.class));
      boolean declaresExceptions = !oneway.getThrownTypes().isEmpty();
      CtParameter<?> OUTorINOUTParams = hasOUTorINOUTParams(oneway);

      if (!isVoid) {
        ValidationPoint.report(Severity.ERROR, oneway,
          "Oneway operations must have no return value");
      }

      if (declaresExceptions) {
       //report error
      }

      if (OUTorINOUTParams != null) {
        //report error
      }

  }

}
```

From this, we can see that checks are performed on the method's signature. It may be that several similar checks are made throughout the AttDSL. If this is the case, a (or multiple) generic meta-annotation(s) will be preferred to a specific one.

5.3 Fraclet

Fraclet is an annotation framework for the Fractal component model [1]. The Fractal component model defines the notions of *component*, *component interface*, and *binding* between components. Each of these main notions is reflected in the AttDSL defined by Fraclet. There are two implementations of Fraclet [17], one using XDoclet2, and the other one using Java5 annotations and Spoon annotation processor. The annotations defined by Fraclet/Spoon are summarized in table 3.

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

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 3: Overview of Fraclet annotations

`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 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[5]. These last three annotations follow the same rules than those of `FractalItf`.

[5]One could argue that the cardinality and contingency elements would be better represented as `enums`; indeed this is the case in new versions of Fraclet. However, the version we reviewed them did not use enums.

```
@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";

}
```

```
@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";
}
```

`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. Meta-annotations are provided to check all these rules.

`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 meta-annotation `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 "";
}
```

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

`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`. Meta-annotations for these rules are included in the definition of the annotation.

`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()` annotation element 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();
}
```

All of the restrictions are translated into AVal annotations. With respect to the XDoclet implementation of Fraclet, our approach both makes explicit the restrictions and enforces them. If a restriction is not met on the XDoclet implementation, for example a class is not marked with `FractalComponent`, but its fields are annotated with `FractalAC`, the expected code is not generated, leaving the programmer to wonder where the problem is located.

6 Related Work

Annotation Engines Previous to the introduction of annotations in Java, XDoclet [24] relied on modified javadoc comments called tags to specify metadata for program elements. In XDoclet2, a form of tag validation is realized by tagging the tag definitions. The set of validations is fixed, and no special facilities are provided for extending them, in contrast with our approach.

Mezini et. al. [2] propose a single meta annotation for the custom attribute facility in the .Net framework. However, they concentrate on *dependencies* between annotations 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 principle of @OP which strives to reduce the use of external configuration files as much as possible.

The Annotation Processing Tool (`apt`) is a command-line tool provided by Sun for the processing of Java annotations. The approach to annotation validation proposed in this article could be implemented in `apt` for the most part, since it provides a reflective API to reason about the annotated program. However, in contrast to our current annotation processing tool Spoon, `apt`'s API only models the declaration of Java elements (classes, methods, interfaces, packages) and does not provide a representation for code *inside* of methods. This is an important limitation because certain types of validations, such as the one given Section 3.4, require the access to the entire AST. This limitation also precludes annotation processing of local variables (which is allowed by the

Java Language Specification [8]). Finally `apt` does not provide any special or generic features that would be similar to our meta-annotations for annotation validation. Annotation validation with `apt` has to be manually programmed by the developer, which is a break to productivity and reuse of commonly used validations.

Active Libraries Also known as semantically enhanced libraries, or library-level optimizations, *Active Libraries* [22] take an active role in interacting with programming tools. Such interaction could, for example, instruct the compiler to check for certain unwanted idioms, or could take an active role in transforming (optimizing) the program that uses the library. Active libraries have been applied in the context of specific domains, such as scientific computing [21, 7], and several *generic* active library definition frameworks, Broadway [9] and Pivot [20] have been proposed. These frameworks provide tools and abstractions so that the library programmer can make its library *active*. In this regard, AVal aids the annotation library (AttDSL) programmer to specify how the compiler should check the annotations in the base program. It is important to note that AttDSLs are eminently active, since the annotations themselves carry no semantic, one must be provided via program transformation, generation or interpretation.

Static Validation Static validators allow developers to check properties of their code that go beyond of that what is provided by normal compilers. Lint [12] 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 meta-annotations.

In [10], 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. In addition to this, Hedin's proposal uses attributes as means to validation, whereas AVal uses them both as means and as subject.

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 [19]. Nevertheless, in an Aspect Oriented language such as AspectJ[11], no extra reflection facilities are provided, so the validation programmer must rely only on Java reflection which does not reify the body of methods. Furthermore, 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.

7 Conclusion and Future Work

The use of meta-annotations for the validation of AttDSLs, such as the one implemented in AVal, provides several advantages. First, it permits the @OP framework developer to express in a declarative way the validations needed to assure the correct usage of the framework, while separating annotation validation concerns from annotation interpretation concerns in the framework. Second, it provides the @OP framework user early checking of the use of the AttDSL, which is checked at compile-time. Third, it eases the adoption of an alien AttDSL by explicitly encoding the restriction of use in the declaration of the annotations themselves, making the annotations an additional source of documentation for the @OP framework.

In this article, we define a meta-AttDSL consisting of seven meta-annotations that can be combined together to express domain-specific validations of some AttDSL rules. With AVal, we provide a Java implementation of this meta-AttDSL, which is straightforward to extend by defining new meta-annotations and their associated validations. We also provide as case studies three @OP frameworks: Saxpoon, Fraclet and the JSR181 for web services, and shown how to use AVal to include syntactic as well as semantic checks in them. We show that AVal is declarative, expressive, and extensible enough to be applied to real-world @OP frameworks.

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). 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.

References

[1] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, 36(11‐12):1257–1284, 2006.

[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] M. Frigo and S. G. Johnson. FFTW: The fastest fourier transform in the west. Technical Report MIT-LCS-TR-728, MIT LCS, 1997.

[8] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, third edition, May 2005.

[9] S. Z. Guyer and C. Lin. Broadway: A compiler for exploring the domain-specific semantics of software libraries. In *Proceedings of the IEEE*, 2004.

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

[11] 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.

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

[13] J. Kort and R. Lämmel. Parse-tree annotations meet re-engineering concerns. In *SCAM*, pages 161–. IEEE Computer Society, 2003.

[14] W. S. Means and M. A. Bodie. *The Book of SAX: The Simple API for XML*. No StarchPress, 2002.

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

[16] R. Pawlak, C. Noguera, and N. Petitprez. Spoon: Program analysis and transformation in java. Technical Report 5901, INRIA, may 2006.

[17] 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.

[18] M. P. A. Sellink and C. Verhoef. Scaffolding for software renovation. In *CSMR*, pages 161–172, 2000.

[19] 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.

[20] B. Stroustrup. A rationale for semantically enhanced library languages. In *Library-Centric Software Design LCSD'05*, Oct. 2005.

[21] T. L. Veldhuizen. Arrays in Blitz++. In *Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, Berlin, Heidelberg, New York, Tokyo, 1998. Springer-Verlag.

[22] T. L. Veldhuizen and D. Gannon. Active libraries: Rethinking the roles of compilers and libraries. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98)*, Philadelphia, PA, USA, 1998. SIAM.

[23] 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.

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

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