Annotation Framework Validation using Domain Models

Carlos Noguera, Laurence Duchien

Annotation Framework Validation using Domain Models

Carlos Noguera and Laurence Duchien

Université des Sciences et Technologies de Lille, INRIA Lille Nord-Europe - LIFL
40, avenue Halley,
59655 Villeneuve d'Ascq, France
{noguera|laurence.duchien}@lifl.fr

Abstract. Frameworks and libraries that use annotations are becoming popular. However, there is not enough software engineering support for annotation development. In particular, the validation of constraints in the use of annotations requires further support. In this paper we postulate that annotation frameworks are a projection of the domain model into a programming language model. Using this idea, we have developed a tool that allows the annotation programmer to specify, and then validate the constraints of the annotation framework regarding a given annotated application using a domain model. To validate our approach to the validation of constraints using models, we apply it to the Fraclet annotation framework and compare it to the previous implementation.

1 Introduction

Large annotation frameworks [6, 16] are becoming more and more common. These kinds of frameworks, such as EJB3 [12], offer to the application programmer, in addition to classes and methods, annotations that provide an additional, declarative way to use the framework. Annotation framework design and implementation rises a number of challenges. Among them, the problem of validating that the application developers correctly use the annotations. It is interesting for the framework programmer to be able to express the constraints of its annotation framework, and to automatically check whether a program is valid with respect to these constraints.

Previously, we have developed a technique and a tool to express these constraints, called AVal, that relies on meta-annotations. Constraints are then implemented by the use of a meta-annotation framework, and are checked by an annotation processor. In this paper, we extend AVal by investigating the relationship between annotation frameworks and domain models. Based on this relationship, we show that the constraints of an annotation framework can be translated into constraints on a domain model. Furthermore, the validation of an annotated program corresponds to the validation of the instance of the domain model. We apply this technique to a case studie: Fraclet.

The paper is organized as follows: first, in the next section we motivate the need for annotation validation, and present our existing approach called

AVal. Then, in Section 3 we present our proposal, by discussing the relation between annotations and models, and their usefulness in annotation validation. In Section 5 we present how models aid in the validation of a real-life annotation framework, Fraclet. Finally, in Sections 6 and 7 we compare our work to similar approaches, and conclude.

2 Annotation Validation

The Java type system for annotations is not expressive enough to assure that the use of annotations is correct. This type system allows the annotation framework developer to define the names, types and default values of (optional) properties, as well as the Java program elements to which it can be attached. It, however, leaves the responsibility of more complex checks to the annotation framework developer.

Complex annotation frameworks impose further restrictions on the use of annotations than those made available by the Java compiler. For example, in EJB3, the `@Id` annotation that marks a field in an entity class as its identifier, can only be placed in fields belonging to a class annotated as `@Entity`. Constraints such as these are common among annotation framework specifications. These kinds of constraints cannot be enforced by the Java compiler, and it is up to the annotation developer to check them as part of the annotation's processing phase.

Annotation frameworks imply a number of constraints on their usage. This is not different than for any other framework. However, in contrast to regular frameworks, annotation frameworks are static entities; that is, their usage can be checked during the compilation of the program. This is done so that the errors are provided to the final developer as soon as possible. Given the static nature of the semantics of annotations, their constraint checking is considerably easier than that of its regular counterparts because, in general, no complex static analysis must be performed.

We call the process of constraint checking *validation of an annotated program*. This process takes as inputs the set of annotation types and a program carrying the corresponding annotations. As output, a set of errors corresponding to the violations of the constraints as they are used in the program are returned. In this process we identify two actors: the developer of the annotation framework, i.e., the person that implemented the annotation types; and the program developer, i.e., the person that wrote and annotated the program.

Although the process flow for the validation of an annotated program is straightforward, the constraints actually checked strongly depend on the particular annotation framework. Each annotation framework imposes its particular set of constraints that derive from the domain in which they lay. In general, annotation validations are of two kinds, those dealing with the relationship between an annotation type and the code element on which it is placed, and those dealing the annotation type's properties, and its relationship with other annotation types. We call the former *code-wise validations*, while the later, *structural validations*.

2.1 AVal - Annotation-based Validation

To perform the validation of annotation frameworks AVal [13] applies the concept of annotations itself by defining an annotation framework that contains a set of meta-annotations for the domain of *annotation constraints validation*. These validation meta-annotations are used to augment the definition of the annotation framework under development with meta-data relevant to validating a given constraint.

Annotating annotations with other annotations has the advantage to make the constraints *explicit* in the annotation framework's definition and *local* to the annotation they apply to. AVal provides a number of built-in validations, and means to add custom ones. It uses Spoon [14] for compile-time reflection and annotation processing, and through it, provides integration to the Eclipse IDE.

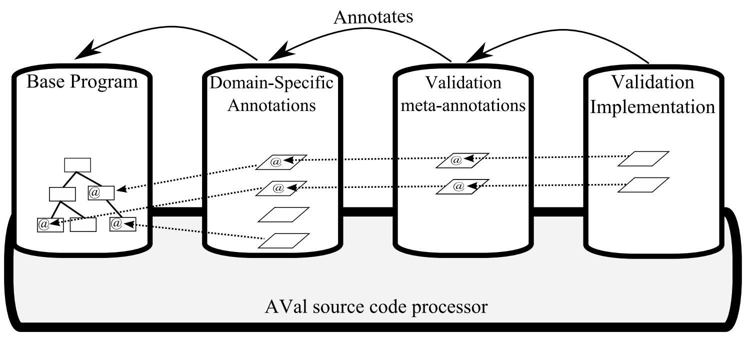

Annotates
Base Program
Domain-Specific Annotations
Validation meta-annotations
Validation Implementation
AVal source code processor

Fig. 1. AVal Architecture

2.2 Annotation Validation for Java

The concept of using meta-annotations to declare restrictions on use of Java annotations is already included in the JDK. Indeed, the Java Language Specification [10] 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.

AVal's architecture is composed of four layers (Figure 1):

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

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

Validation meta-annotations: AVal annotations that encode the constraints to validate domain specific annotations. Each meta-annotation represents a validation constraint, 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, annotation framework, 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 annotation 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. In order to ease the specification of constraints, AVal provides a number of annotations that represent commonly used rules. A subset of these annotations, as well as their description is shown in Table 1, for a more complete discussion on AVal's annotations see [13].

Annotation	Description
`Inside(`AT`)`	The annotation must be placed on elements which are inside elements annotated with AT
`Requires(`AT`)`	The annotation requires that the target of the annotation also is annotated with AT
`RefersTo(`AT`,`N`)`	The property of the annotation must carry the same value as the property called N belonging to AT
`AValTarget(TE)`	The annotation must have as target the program element TE

Table 1. Default constraint annotations in AVal

In order to validate complex constraints, the annotation framework developer can extend AVal with new meta-annotations. For this, a new annotation type and its corresponding implementation (see Figure 1) must be provided. The annotation type serves to mark the context of the constraint, while the actual checking must be performed by traversing the AST of the program. This traversal, of course, requires an intimate knowledge of the model used by the tool to represent the program, and its associated API. Hence, the creation of new annotations for AVal can be a tedious task. In order to ease the extension of AVal's annotations, an abstraction over the AST of the program is needed. Annotation models is one possible abstraction.

3 Annotation Models

As we have seen in the previous sections, complex annotation frameworks require validations that concern both other annotations and the program on which they are used. But, where do these constraints come from? Consider the `Inside` validation, if an annotation type `A` is required to reside inside another one, `B`, this implies a relationship between them since it makes no sense for `A` to be present in the program without its corresponding `B`. Now, suppose that both `A` and `B` are classes in an UML class model, then the relationship induced by the `Inside` validation could be described by means of a containment association between them.

3.1 Annotation and Code Models

Extending this idea of *modeling* annotation types, we can see that structural validations can be mapped to relationships and invariants on a model that represents the annotation types. We will call this model derived from the annotation types an *annotation model*.

Code-wise validations, on the contrary, cannot be described in terms of the annotation model alone, since they deal with constraints on the relationship between the annotations and the program on which they are imposed. To integrate code-wise validations into the model we need a representation of the target language, in this case Java, so that an association between annotation models and code models can be reasoned on. This model representation of the target language, called *code model*, needs to be at the same level of abstraction as that of the annotation model to be able to mark the references from one to the other, i.e. to state that a given annotation is to be placed on a given code element. This code model is the AST of the language.

In this way, the annotation model corresponds to the model of the domain, defined by the annotation types and their corresponding constraints, while the code-wise validations define a *mapping* between the annotation model and the code model. Now, just as the annotations in the program conform to their annotation types, there is a model instance for both the annotation model and the code model. Code-model instances would represent the concrete syntax tree of a given program; while annotation-model instances would represent the annotations present in the target program.

3.2 Example

To better understand the nature of the annotation models and their interaction with the code model, let us suppose a simple annotation framework to define SAX-based XML parsers called SAXSpoon. SAX frameworks traverse the tags of an XML document up-calling a method each time a start or end tag is found. In our annotation framework, we will define three annotations:

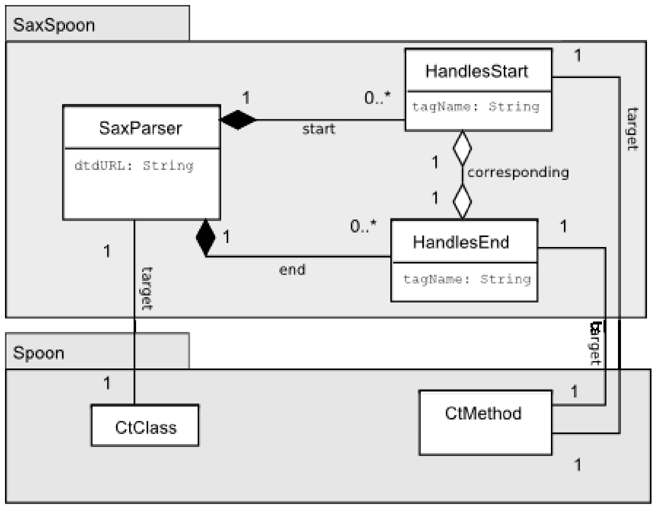


Fig. 2. SAXSpoon annotation model

`SAXParser`, `HandlesStart` and `HandlesEnd`. `SAXParser` identifies a class as being a SAX parser and it defines a single property that points to the DTD document that defines the type of documents that the class will handle. `HandlesStart` and `HandlesEnd` respectively identify the methods that handle the start and end of a tag, given as parameter to each annotation. The corresponding annotation model for this framework is depicted in Figure 2. In it, The package `SaxSpoon` contains the annotation model, while the package `Spoon` contains the relevant parts of the code model.

Code-wise constraints for SAXSpoon can be encoded as OCL expressions on the relations between the `SaxSpoon` and `Spoon` packages; while structural constraints can be encoded in the relationships between the elements of the annotation model itself.

4 Validation using Annotation Models

As discussed before, it is possible to embed constraints on the annotation and code model. These constraints are then checked against a model instance derived from an annotated program. In order to do this, the annotation developer must be able to declare the constraints on its annotation framework, and she must be able to direct the way in which the annotation model instance is generated. For this, we have extended AVal (as presented in Section 2.1) with annotations to specify the instanciation of the model (`Association` and `DefaultValue`) as well as the constraints on the model (`OCLConstraint`). The implementation for the aforementioned annotations and its corresponding tool chain is called *ModelAn*, and will be explained in the following Section.

4.1 ModelAn - Model Based Annotation Validation

ModelAn is a tool chain for the definition of annotation model constraints and their corresponding validation. It is driven by annotations (as opposed to models), and it uses AVal as an underlying layer. The workflow for the use of ModelAn is depicted in Figure 3. It starts from the annotation types defined as part of the framework by the annotation framework developer. The set of annotation types carry annotations that direct the `Model Extraction` engine in producing the annotation model and its corresponding `Model Instanciator`. The program written by the application developer is then fed to it, producing an annotation model instance that conforms to the annotation model extracted before. Using both the model and its instance, ModelAn uses a constraint checker to validate the program, and report back to the application developer any violations. The whole process is transparent to the application developer.

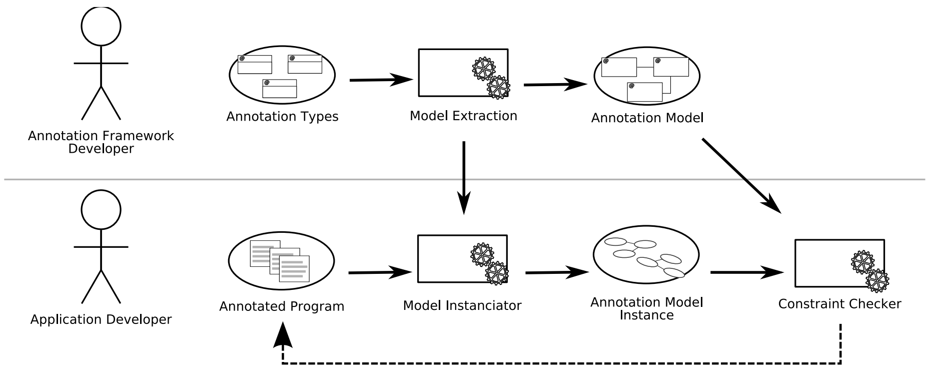


Fig. 3. ModelAn process flow

Model Extraction

The annotation model is extracted from the set of annotation types that compose the framework. As a starting point, each annotation type is represented as an element of the model with its corresponding attributes. In addition to this, each element in the model is associated with the code element which it is supposed to augment. This association is called the *target* of the annotation.

The model is then augmented by the annotation framework developer using two annotations on the annotation types: `Association` and `DefaultValue`

Association Associations define the structural relations between annotations. An association must define a `name`, a `type` and a defining `query`. The OCL query is evaluated in the context of the annotation type on which it is placed, and can only reason on associations on the code model because it itself is defining the associations on the annotation model. For example, in the SAXSpoon annotation framework, there is a relation between a SaxParser and its start and end handlers. Therefore, the definition of the SaxParser annotation type would be as follows:

```
@Association(name = "Start",
             type = HandlesStart.class,
             query =
"HandlesStart.allInstances()->select(self.target.Methods->includes(target))")
public @interface SaxParser {
        String dtdURL() default "";
}
```

In this example, the query traverses all the `HandlesStart` elements, looking for those which are placed on methods which belong to the class annotated with `SaxParser`. Hence, this query constructively defines the relation *start*. A similar construction is used to define the relation between `SaxParser` and `HandlesEnd`

DefaultValue Attributes in annotations often have default values. In the general case, the default value is a static value (for example the empty string), but in some cases, the default value depends on the place in which an annotation is placed. For example, suppose that the name of the tag that a method handles is by default the name of the method. In this case, the default value cannot be known when the annotation type is defined, since it will change depending on the use of the annotation. The annotation framework developer can then state, using an OCL query, what the default value of the property should be. In the case of SaxSpoon, the definition of the `HandlesStart` would be:

```
public @interface HandlesStart{
  @DefaultValue("self.target.SimpleName")
  String tagName();

}
```

Model constraint definition

Once the annotation model has been defined, the developer can define the constraints on it. In order to do this, ModelAn defines a single annotation, `OCLConstraint` that is to be placed on the annotation type. The constraint is represented by an OCL expression that is evaluated in the context of the annotation model element that corresponds to the current annotation type.

OCLConstraint OCL expressions placed on annotation types can use the associations defined by the `Association` annotation to express the constraints of the annotation framework. The `OCLConstraint` annotation is an AVal annotation (see Section 2.2) that defines a single property that contains the expression itself. In SaxSpoon, the annotation framework developer may want to specify a constraint stating that a warning should be raised if a Sax parser handles the Start, but not the end of a given tag. For this, a constraint must be placed in the *corresponds* relation:

```
@Association(name = "corresponds",
             type = HandlesEnd.class,
             query =
"HandlesEnd.allInstances()->select(handler|handler.tagName = self.tagName)")
@OCLConstraint("self.corresponds->size() = 1")
public @interface HandlesStart {
        String tagName();
}
```

In this example, a *corresponds* association is defined using the first `Association` annotation, and the second `OCLConstraint` annotation places an OCL constraint that uses it to specify that there should be a single corresponding tag handler for the same tag.

Using the information defined by the `Association`, `DefaultValue` and `OCLConstraint` annotations, the model extraction engine generates an Ecore file that contains the annotation model. The Ecore annotation model references the SpoonEMF [2] Ecore model that represents the Java programming language, i.e. the code model. The resulting annotation model for our running example is shown in Figure 4. The OCL queries that define the associations in the model are saved in this file by means of ecore-annotations on the references, while an annotation on the element states the annotation type that this element models. The OCL constraints are not included in the model itself.

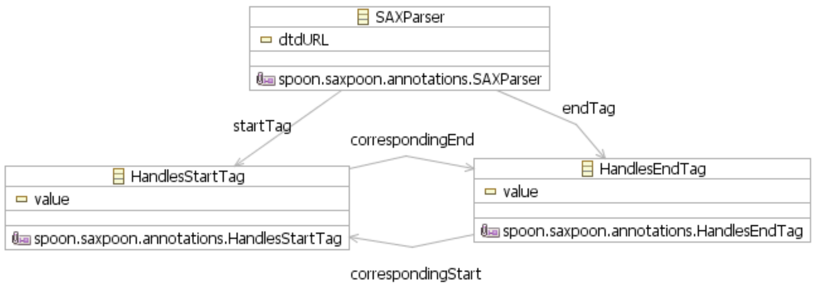
Finally, the model extraction engine will generate a set of Spoon source code processors that will instantiate the annotation model and the code model of a given program.

Model Instanciator

The annotation model is instantiated by a source code processor generated by the model extraction engine. The source code processor traverses the AST of the target program, and using the Ecore file that contains the annotation model, creates an instance of the corresponding element for each annotation it encounters. Once all the annotations in the program have their corresponding instance, the source code processor executes the OCL queries present in each association in order to populate them. At the end of the process, an in-memory instance of the annotation model that represents the annotated program is available. This instance can then be used to check the annotation framework constraints.

Constraint Checker

The constraints themselves are checked using AVal. The AVal source code processor passes over the program after the model instantiator. Each time an annotation with an OCL constraint is found, the OCL expression is evaluated on the model's instance. If the expression evaluates to `false`, an error is raised. Since the constraint checker uses AVal, the errors are presented to the programmer in the same format as Java compiler errors (see [13]).


SAXParser
dtdURL
spoon.saxpoon.annotations.SAXParser
startTag
endTag
correspondingEnd
HandlesStartTag
value
spoon.saxpoon.annotations.HandlesStartTag
HandlesEndTag
value
spoon.saxpoon.annotations.HandlesEndTag
correspondingStart

```
@Associations({
@Association(name="startTag",
 type = HandlesStartTag.class,
 query= "HandlesStartTag.allInstances()->"+
        "select(self.target.Methods->includes(target))"),
@Association(name="endTag",
 type = HandlesEndTag.class,
 query= "HandlesEndTag.allInstances()->"+
        "select(self.target.Methods->includes(target))"),
})
public @interface SAXParser {
        String dtdURL() default "";
}

@Associations({
@Association(name = "correspondingEnd",
 type = HandlesEndTag.class,
 query = "HandlesEndTag.allInstances()->"+"
    select(handler|handler.value = self.value)")
})
@OCLConstraint("self.correspondingEnd->size() = 1")
public @interface HandlesStartTag {
        String value();
}

@Associations({
@Association(name = "correspondingStart",
 type = HandlesStartTag.class,
 query = "HandlesStartTag.allInstances()->"+
    "select(handler|handler.value = self.value)")
})
@OCLConstraint("self.correspondingStart->size() = 1")
public @interface HandlesEndTag {
        String value();
}
```

Fig. 4. SAXspoon Ecore Model and Annotated types

Although the process to validate the constraints of a program starting from the annotation framework, all the way to obtaining the errors may seem long, it is important to note that each of the actors see only a one-step process. Indeed, for the annotation framework developer, only the model extraction step is necessary, while for the application developer, the model instantiation and constraint checking are a step of the compilation process, and therefore transparent. In terms of advantages, the use of models to define the constraints of an annotation framework allows the annotation framework developer to abstract away from the code model, and reason about the relations in the domain model itself. In the SaxSpoon example, this is evident in the constraint that establishes the correspondence between start and end handlers. In this constraint, the annotation framework developer does not refer to any code element in the constraint's expression, reasoning instead only on the actual domain of the annotation framework. It is also important to note that this additional abstraction level comes at no cost to the final application developer, since as we pointed out before, the constraint checking is hidden behind the compilation of the program. The framework developer, in contrast, is required to manipulate OCL expressions, which can be complex at times, to define the associations and constraints of the annotation framework. In order to reduce the use of OCL we expect to leverage UML's stereotypes as a way to graphically specify the annotation model. This is further discussed in Sections 6 and 7.

5 Case Study - Fraclet

Annotation	**Location**	**Parameter**	**Description**
`Component`	Class	*name*	Annotation to describe a Fractal component.
`Interface`	Interface	*name, signature*	Annotation to describe a Fractal business interface.
`Attribute`	Field	*argument, value*	Annotation to describe an attribute of a Fractal component.
`Required`	Field	*name, cardinality, contingency*	Annotation to describe a binding of a Fractal component.
`Lifecycle`	Method	*value*	Annotation that marks a method as a life-cycle callback. The parameter specifies the step of the life-cycle.
`Controller`	Field	*value*	Annotation that marks a field as an access point to the component's reflective services

Table 2. Overview of Fraclet annotations

Fraclet is an annotation framework for the Fractal component model [3]. The Fractal component model defines the notions of *component*, *component interface*, and *binding* between components. Each of these main notions is reflected in the annotation framework defined by Fraclet. There are two implementations of Fraclet [15], one using XDoclet2, and the other one using Java5 annotations and Spoon annotation processors. The annotations defined by Fraclet/Spoon are summarized in Table 2.

In Figure 5, Fraclet/Spoon is used to augment a Java class in order to represent a Fractal primitive component. The `Client` class uses a `Component` annotation to represent a component called `helloworld.Client` that provides a single interface named `r`. Fields of this class are marked as attributes, required ports or controller hooks. Finally, a method on the component is marked as a life-cycle handler.

```
@Component(name = "helloworld.Client",
           provides = @Interface(name = "r",
                                 signature = Runnable.class))
public class Client implements Runnable {

 private final Logger log = getLogger("client");

 @Attribute(value="Hello world") private String message;
 @Requires(name="s")             private Service service;
 @Controller("name-controller")  protected NameController nc;

 @Lifecycle(CREATE) protected void whenCreated() {
   log.info("helloworld.Client - created.");
 }

 public void run() {
   this.service.print(this.message);
 }
}
```

Fig. 5. Client Comoponent Fraclet Implementation

In order to define the constraints of each of these annotations, we have applied the meta-annotations defined in Section 4.1 to extract an annotation model that represents Fraclet. The resulting model is shown in the Figure 6.

Once, the model was defined, we discussed with the Fraclet developers in order to learn the constraints that a correct Fraclet application must adhere to. Then we translated the constraints to their corresponding OCL expressions. These are the constraints and their corresponding translations:

A Component's name must be unique in the application By default, the name of a component is the simple name of the class on which the `Component`

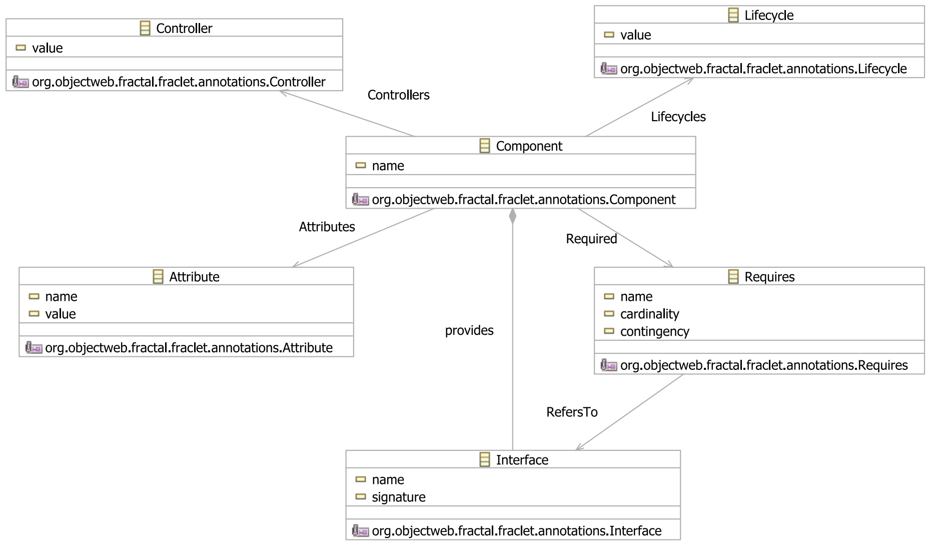

Controller
value
org.objectweb.fractal.fraclet.annotations.Controller
Lifecycle
value
org.objectweb.fractal.fraclet.annotations.Lifecycle
Controllers
Lifecycles
Component
name
org.objectweb.fractal.fraclet.annotations.Component
Attributes
Required
Attribute
name
value
org.objectweb.fractal.fraclet.annotations.Attribute
provides
Requires
name
cardinality
contingency
org.objectweb.fractal.fraclet.annotations.Requires
RefersTo
Interface
name
signature
org.objectweb.fractal.fraclet.annotations.Interface

Fig. 6. Fraclet Annotation Model

annotation is placed. This is expressed using a `DefaultValue` annotation. The definition of the component annotation is as follows:

```
public @interface Component {
 @Default("self.target.SimpleName")
 @OCLConstraint("Component.allInstances()->"+
   "select(c:Component| c <> self and c.name = self.name)->isEmpty()")
 String name() default "";
//...
}
```

A Field on a Component cannot be at the same time Attribute and Required

```
@OCLConstraint("Requires.allInstances()->"+
 "forAll(r:Requires|r.target <> self.target)")
public @interface Attribute {
//...
}
```

A Required Interface must be defined. The name of the required interface is by default the name of the field on which it is placed.

```
@OCLConstraint("self.RefersTo->size() = 1")
@Association(type = Interface.class, name="RefersTo",
  query="Interface.allInstances()->select(i|i.name = self.name)")
public @interface Requires {
    @Default("self.target.SimpleName") String name() default "";
}
```

The previous version of Fraclet (studied in [13]) used AVal and needed six different annotations to perform the same tests we have implemented here with only `OCLConstraint`. In addition to this, we were able to elegantly specify the default values for the names of the `Component` and `Attribute` annotations, which was not addressed in the previous AVal-based version.

6 Related Work

Related work in annotation framework validation and development can be aligned along two axes: the development and validation of annotation frameworks and the relationship between annotations and models.

Annotation Validation The need of validating annotations has been previously addressed by academia. First, in [5] Cepa et. al. propose a mechanism to validate the use of custom attributes (similar to annotations in the .NET platform) by placing custom attributes on the definition of other custom attributes. This is quite similar to the approach proposed in AVal, however, their technique only allows for the definition of structural constraints, and no mechanism to extend the constraints is provided. Also, they provide no explicit code or annotation model. In [7], Eichberg et.al. propose to validate structural annotation constraints in Java programs by representing a program as an XML document, and representing the constraints as XPath queries. In their approach, the XML schema of the document acts as an implicit code model, but they do not provide an explicit annotation model. The lack of this model complicates the definition of constraints, since no relation exists between annotations.

Finally, annotations are extensively used for the validation of programs [11, 8, 9]. However, this use of annotations differs from our intention, since our focus is the validation of the use of annotations themselves, not of the program on which they are placed.

Annotations and Models Annotations, seen as meta-data attached to a code entity, are semantically close to stereotypes as defined in UML 2.0 [1]. Indeed, it is common to represent annotations, during design, as stereotypes [4]. Nevertheless, it is difficult to establish a direct mapping between stereotypes and annotations given the particularities of annotations. For exmaple, annotations do not allow for inheritance, an annotation can be placed on different code elements (stereotypes are restricted to one), and most importantly, annotations can refer to types that are defined in the program in which they are applied since for example, annotations can contain as a property `enums` defined in the program. This last characteristic is the most problematic, since it places annotation models somewhere in between levels M1 and M2. Nevertheless, it seems possible to construct a mapping between UML profiles and annotation models. This will be the subject of future work.

The use of models for the development of annotation-based programs is explored in [16] by Wada et. al. They propose a full MDA approach that starts from a model, and ends with an executable program. However, they start from the idea that the annotation framework already exists, and therefore, provide no support for the development of it. We believe their proposal and ours to be complementary.

7 Conclusion

We presented a way to specify and validate constraints on annotation frameworks based on models. As an implementation, we introduced ModelAn, an annotation framework that allows the annotation framework developer to define an annotation model and attach constraints to it. ModelAn also constructs a source code processor that generates instances of the annotation model in order to validate the constraints. The use of OCL constraints offers the developer a greater degree of expressiveness when defining new kinds of constraints with respect to the extensibility options of AVal. In addition to this, the use of a code model and an annotation model provides an abstraction over the direct manipulation of the AST of the program, which results in more concise constraints.

The use of models to define the constraints of an annotation model allows the annotation framework developer to abstract away from the code model, and reason about the relations in the domain model itself. Using OCL also provides a declarative way to express these constraints, and diminishes the prerequisite knowledge of the underlying AST API that the annotation developer must have. This two characteristics make ModelAn a more extensible annotation framework validation platform than AVal.

As future work, we believe that the relation between annotations and models can be further exploited. In particular, we are working on a model-directed approach that allow us to define an annotation framework from a set of UML stereotypes and their corresponding constraints. This will be done by implementing a model-to-model transformation that goes from a profile to an annotation model, and then to the actual implementation. Also, annotation models can prove to be of aid in the understanding of an annotated program, since they make explicit the relation between annotations on it. Furthermore, if a program uses different annotation frameworks to implement different concerns on different domains, then each associated annotation model will provide a domain specific view of the program.

References

1. *OMG Unified Modelling Language Infrastructure (OMG UML) V.2.1.2*, Nov. 2004. `http://www.omg.org/spec/UML/2.1.2/Infrastructure/PDF`.
2. O. Barais. SpoonEMF, une brique logicielle pour l'utilisation de l'IDM dans le cadre de la réingénierie de programmes Java5. In *Journées sur l'Inénierie Dirigée par les Modèles (IDM)*, June 2006. Poster.
3. 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.
4. V. Cepa and S. Kloppenburg. Representing Explicit Attributes in UML. In *7th International Workshop on Aspect-Oriented Modeling (AOM)*, 2005.
5. 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.

6. A. Cisterino, W. Cazzola, and D. Colombo. Metadata-driven library design. In *Proceedings of Library Centric Software Development Worksshop*, Oct. 2005.
7. 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.
8. 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.
9. D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, Jan./Feb. 2002.
10. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, third edition, May 2005.
11. G. Hedin. Attribute extensions - a technique for enforcing programming conventions. *Nord. J. Comput*, 4(1):93–122, 1997.
12. L. D. Michel and M. Keith. *Enterprise JavaBeans, Version 3.0*. Sun Microsystems, May 2006. JSR-220.
13. C. Noguera and R. Pawlak. AVal: an extensible attribute-oriented programming validator for java. *Journal of Software Maintenance and Evolution*, July 2007.
14. R. Pawlak, C. Noguera, and N. Petitprez. Spoon: Program analysis and transformation in java. Technical Report 5901, INRIA, may 2006.
15. 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.
16. 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.