N° ordre 4280

Département de formation doctorale en informatique
UFR IEEA

École doctorale SPI Lille

Un ensemble d'outils orientés modèle pour développer et valider des cadres logiciels à base d'annotations

THÈSE

présentée et soutenue publiquement le 25 Nov. 2008

pour l'obtention du

Doctorat de l'université des Sciences et Technologies de Lille

(spécialité informatique)

par

Carlos Noguera

Composition du jury

Président :	Gilleron Rémi	Professeur des Universités, Université Lille 3
Rapporteurs :	Consel Charles	Professeur des Universités, ENSEIRB
	Jézéquel Jean-Marc	Professeur des Universités, Université Rennes 1
Examinateurs :	Deridder Dirk	Docteur, Vrije Universiteit Brussel
	Donsez Didier	Professeur des Universités, Université Joseph Fourier
	Pawlak Renaud	Docteur, ISEP
Directrice :	Duchien Laurence	Professeur des Universités, Université Lille1

Laboratoire d'Informatique Fondamentale de Lille – UMR USTL/CNRS 8022 – INRIA Lille Nord Europe

Résumé

Les annotations, dans le langage de programmation Java, sont un moyen d'embarquer des méta-données dans le code source d'un programme. Elles peuvent être utilisées pour étendre le langage Java avec des concepts spécifiques à un domaine. Utilisées de cette manière, les annotations offrent un moyen de réduire le fossé sémantique entre les concepts du domaine et les concepts fournis par le langage de programmation. Pour l'utilisateur de ce cadre d'annotations (c.-à-d., le développeur d'applications), il est important de comprendre comment les différentes annotations se relient entre elles afin de les utiliser correctement et de détecter des erreurs d'utilisation au plus tôt.

Des cadres d'annotation ont déjà été adoptés par l'industrie. Cependant, leur développement demeure complexe et est fait, en grande partie, de façon ad-hoc. En développant un cadre d'annotations, le développeur doit s'assurer que le programme qui emploie les annotations est conforme aux contraintes définies pour elles. Telles contraintes sont souvent reléguées à la documentation du cadre logiciel, puisque les processeurs d'annotation existants ne fournissent pas une façon de les spécifier et vérifier. En plus de ceci, les processeurs d'annotations existants offrent comme entité de manipulation juste l'arbre de syntaxe abstraite (AST) du programme. Ceci force le développeur à réifier les annotations elles-mêmes s'il veut des éléments d'abstraction d'un plus haut niveau.

Pour aider le réalisateur de cadre d'annotations, nous proposons deux cadres d'annotations. Le premier, appelé AVal, fournit un nombre de contraintes réutilisables, déclaratives et extensibles pour spécifier un cadre d'annotation. Les applications annotées peuvent ainsi être interprétées afin de les valider. Le deuxième cadre, appelé ModelAn, permet au réalisateur de cadre d'annotations de décrire un modèle qui correspond aux annotations définies dans cadre d'annotations. Par ce moyen, le réalisateur peut exprimer les contraintes en termes du modèle. Des programmes annotés sont alors représentés comme instances du modèle (d'annotation), et les contraintes sont vérifiées sur cette même instance. A partir de ce modèle d'annotations, les classes Java qui réifient les annotations sont produites. Les annotations réifiéess peuvent alors servir de point de départ à l'interprétation du programme annoté.

Pour valider notre approche, nous avons construit le modèle d'annotation et décrit leurs contraintes à l'aide d'AVal et d'expressions OCL pour trois cadres d'annotation industriels: Fraclet, un cadre d'annotation pour le modèle composant de Fractal, JWS pour le développement de services Web dans Java, et l'API de persistance Java, une partie de la spécification Java de EJB3.

Mots-clés: Programation à base d'annotations, Ingénierie dirigée par les Modèles, Langages Dédies, Validation de programmes

Abstract

Annotations, in the Java programming language, are a way to embed meta-data into the source-code of a program. Annotations can be used to extend the Java language with concepts specific to a domain. When used in this manner, annotations serve as means to reduce the semantic gap between concepts in the problem domain and the concepts provided by the programming language. For the annotation framework user (i.e., the application programmer) it is important understand how different annotations relate to each other in order to correctly use them and to get errors as soon as possible when not.

Annotation frameworks are already being adopted by industry; however, their development remains complex, and it is done largely in an ad-hoc manner. When developing the annotation framework, the programmer must make sure that the program that uses the annotation complies with the constraints defined for it. Such constraints are often relegated to the documentation of the framework, since current annotation processors do not provide a way to specify and check them. In addition to this, current annotation processors just offer to the framework programmer the AST of the program as manipulation entity. This forces the programmer to reify the annotations himself if he wants higher abstraction elements.

To help the annotation framework developer, we propose two annotation frameworks. The first one, called AVal, provides a number of reusable, declarative and extensible constraints that can be used to specify the annotation framework, and can be interpreted in order to validate an annotated program. The second one, called ModelAn, allows the annotation framework developer to describe a model that corresponds to the annotations in the framework, and to express the constraints in terms of this model. Annotated programs are then represented as instances of the (annotation) model, and the constraints checked on it. From this model, Java classes that reify the annotations are generated. The reified annotations can serve as starting point for the interpretation of the annotated program.

To validate the approach, we construct the annotation model, and describe the constraints as AVal and OCL expressions of three industrial annotation frameworks: Fraclet, an annotation framework for the Fractal component model, JWS for the development of web services in Java, and the Java Persistence API, part of the EJB3 specification.

Keywords: Annotation-based programming, Model-Driven Engineering, Domain-Specific Languages, Program Validation

Acknowledgement

I would like to start by thanking the members of the jury, especially my two reporters Charles Consel and Jean-Marc Jézéquel for their time and valuable comments on the document. I would also like to express my gratitude to the examinors, Rémi Gilleron, Dirk Deridder and Didier Donsez. I would like to specially thank Renaud Pawlak, for his participation on my jury, and more importantly for his guidance, advice and friendship during the first half of my PhD. Be it in the lab, developping Spoon or in the "Paon d'Or"; his knowledge, sense of humor and good disposition made my stay here all the more enjoyable.

During this three years I have had the luck of encountering numerous colleges who have since left, and from whom I have leaned much. To all of them I owe my gratitude: Nicolas Petitprez, the original Spoon-man, Dolorès Diaz, Nicolas Pessemier, Guillaume Dufrêne, Naouel Moha, Maja D'Hondt, Ellen Van Paeschen, Johan Brichau, Tom Mens, Michal Malohlava and Jérémy Dubus. Among these temporary office mates, Johan Fabry occupies a special place, for in him I found an inconditional friend.

It goes without saying that I would not have been able to survive these three years without the help and support of the permanents of the team: Anne-Françoise Le Meur who was always there for reading a paper or just talking about life; Phillipe Merle, Lionel Seinturier and Romain Rouvoy to whom I look up to, and who are my models on what a researchers life should be. Finally, and more importantly Laurence Duchien, who more than an advisor, was my guide during these three long years. Laurence was always there to encourage me and she always had time to hear my problems both in the lab and outside.

To the PhD students with whom I shared times both hard and happy, know this: It does end, and it is worth it. All the best of luck to Ales, Guillaume, Carlos, Daniel, Gabriel, Mathiew, Hani, Alban and the other Guillaume; I know that you will be able to get through the PhD and succeed in what ever path you choose to follow afterward. Thanks to the engineers of the team: Nicolas, Pierre, Yann, Damien, and in particular Frédéric, for coffee breaks and small talk (not the language).

In a more personal level, I would like to thank my parents and brothers whose ever present advice and support pushed me through even from a thousand miles away. Finally, I could not have made it through these three years without the unconditional support, advice and love of my wife Angela. With out you, I would not be the man I am today; all I have I owe to you.

A Angela y a Angela

Contents

List of Figures

List of Tables

1

Introduction

Contents

Programming consists of taking concepts of a problem domain and mapping them to concepts offered by a programming language; for example, a *Customer* is mapped to a class. The semantic distance between concepts existing in the problem domain and the ones offered by the programming language constitutes a *semantic gap*. Ways to bridge this gap have been, and still are, actively sought. When developing a programming language one must balance the generality of the concepts offered by the language with how large this semantic gap is. Programming languages that offer concepts too generic are hard to program for, while programming languages with concepts too specific are of limited applicability. For example, in class-based object oriented languages such as Java, classes are used to represent concepts in the problem domain, with fields representing their attributes, and methods the behaivour of those concepts. Although classes strike a good compromise between genericity and specificity, the fact remains that in the mapping of domain-level concepts to classes information is lost. An example of this occurs in Java, where whenever mapping behavior to methods it is imposible to expresss that certain methods modify the state of the concept, while others do not.

One way to deal with this balance is to develop a language that provides support for domain-specific extensions, so that the concepts in the program are generic, but more specific ones can be provided. An example of such extension support is found in Common Lisp's macros [MFK04]. In Java, such extensions are made possible with the inclusion of annotations. Annotations are meta-data attached to program elements signifying that they retain domain-specific semantics. In a manner, they are similar to compilation directives or *Pragmas*, only that annotation's semantics are not defined by the language, but by an external interpretation engine. The fact that annotations are accompanied of an interpretation engine makes them closer to frameworks than to pre-processor directives or macros. A framework is defined by Johnson [Joh97] as "a reusable design of all of parts of a system that is represented as a set of abstract classes and the way their instances

interact". In the case of annotation frameworks, instead of extending abstract classes, the application developer annotates, and the way the annotations are interpreted is similar to the computations resulting from the interaction of instances of the framework.

The use of meta-data attached to program elements on an application is being actively applied in several frameworks both large and small [RPPM06, MK06, Zot05, BK06]. Nevertheless, their development is still largely performed in an ad-hoc manner. Annotations as defined in the Java language [GJSB05] allow the developer to tag certain elements of a program (classes, packages, methods, etc.) to declare that they maintain domain-specific semantics. The use of annotations provides a number of advantages: First, it provides an additional interaction mechanism between an application and the framework that provides the annotations. Second, it enhances the decoupling between the interface of the framework (represented as annotations) and its implementation (the interpretation of the annotations). This decoupling makes annotations attractive for the development of specifications that are to be implemented by third parties; for example the Java Persistence API (a part of the EJB3 [MK06] specification), or the Web Services Meta-data specification for Java [Zot05]. Third, since annotated elements declare that those elements retain a domain-specific meaning in addition to the one provided by the language, annotations can be used to extend the Java language, as is the case in AspectJ5 [KHH+01][1].

The different components of an annotation framework, as well as the challenges in developing them are discussed in the next section 1.1, while our proposal to address the identified challenges is outlined in section 1.2. Finally, an overview of the rest of this document is presented in section 1.3.

1.1 Annotation framework development

We define an annotation framework as a framework [Joh97] that uses as means of interaction, annotations. They are comprised of two main parts: the set of annotation types and the interpretation engine. The annotation types are the interface of the framework. They represent concepts that the user of the framework needs to extend with his own in order to use the services offered by the framework. Annotations in Java are types similar to interfaces, which contain a number of *annotation elements*. They function as read-only fields in classes. The annotation user is in charge of mapping the concepts of his application (represented as classes, packages, methods, fields) to the concepts provided by the framework. Such mapping is done explicitly by decorating the source code of the mapped element with an annotation. When doing this, the annotation user must be aware of the semantics of the annotation types he is using, and in turn, the annotation framework must give timely errors and warnings whenever the developer violates the constraints of the annotations.

In addition to the challenge of checking and specifying the semantics of the annotation types, the annotation framework developer must overcome other challenges when designing and implementing the framework. The annotation types, as already said, represent a number of concepts which stem from the domain the framework represents. It is from this domain-model that some of the constraints of the annotation types originate.

[1] AspectJ – `http://www.eclipse.org/aspectj/`

Now, in designing the annotation types for the framework, the developer is faced with the limitations imposed by the programming language in the definition of annotation types: Annotations can only define a limited number of types for their elements, inheritance between annotations is not supported, nor are associations between annotations. This constitutes challenge `I` in the development of annotation frameworks. Challenge `III` is that of defining and checking annotation constraints. Two kinds of constraints exist for annotations: those that define the relations between annotation types, and those that define the relations between annotations and the source-code elements on which they are placed. The first kind of constraints is rooted on the domain-model that the annotations represent; while the second one comes from the mapping of the domain-model onto the code-model (AST) of the Java language. It is in defining this mapping that challenge `II` is found.

Finally, challenge `IV` deals with the interpretation of annotated programs. Annotation types in Java have no defined semantics. The behavior of an annotation is given by the interpretation performed by the annotation framework. Annotated programs can be interpreted either at compilation-time, where the source code of the annotated program is transformed to an un-annotated one; or at runtime, where using the Java reflection API, the computations of the program are changed as directed by the annotations present. In either case, to interpret annotations it is desirable to reify them so that they resemble their originating domain models. Currently, no annotation processing tool offers such reification, leaving the task to the annotation framework programmer.

To summarize, the challenges identified in the development of annotation frameworks are:

I The representation of domain concepts as annotation types

II The mapping of annotation types to code elements

III The definition of constraints to validate annotated programs

IV The reification of annotations for their interpretation.

1.2 Proposal

We propose to attack the challenges in annotation framework development in two steps. First, by analyzing the constraints in existing frameworks, and from them extracting a number of *generic constraints*. These constraints can then be parametrized by framework developers to specify their annotation types. Second, by borrowing concepts from the model-driven engineering domain to create *annotation models* that are of a higher abstraction level than plain annotation types. Annotation models enable the definition of the mapping of annotation types to code elements and provide support for the reification of annotations in applications. The implementation of both approaches is based on the Spoon[PNP06, Paw05] source code transformation framework. Spoon is particularly suited for the analysis and transformation of annotations, since it supports Java5 syntax, and provides special processors[2] for annotated elements.

[2]Visitors of the AST of the program

Generic Constraints We classify constraints in annotation frameworks in two large groups: Annotation-wise constraints, which deal with constraints on the possible values for the elements of annotations, and on the relationships between annotations; and Code-wise constraints, that deal with the properties that the code elements require in order to be annotated with a particular annotation type. For each group we define a number of constraints, and we define how they are composed in order to specify an annotation type.

In order to be able to use generic constraints, we have implemented an annotation framework for the domain of annotation constraint definition and validation called AVal[3] [NP07, NP06]. Each of the generic constraints is represented by an annotation type. The annotation framework developer uses AVal by putting the constraint annotation on the annotation types of his framework. AVal relies on the Spoon framework to check the constraints on application programs. This check is done over an AST representation of the program provided by Spoon.

AVal annotations provide a generic, reusable and declarative way to specify the constraints of an annotation framework. In addition to this, AVal also provides a way to check that annotated programs conform to the specification of the annotation frameworks that they use. Generic constraints and AVal are further discussed in chapter 4.

Annotation Models When designing annotation types, the framework developer must cope with the restrictions the Java language imposes. Foremost among these restrictions is the one that prevents annotation types from defining relations with other annotation types. Relationships between annotations originate on the domain model that they represent, such relations are often necessary to define constraints, and to interpret annotated applications. Because of this, we propose to augment annotation types with the notion of association. As with generic constraints, we implement this extension through an annotation framework called ModelAn[4] [ND08]. ModelAn defines an `Association` meta-annotation that, when placed on an annotation type, represents an association with another annotation type. The resulting graph of annotation types and the associations between them is what we call an *annotation model*.

Since annotation models contain information that is closer to the one existing in the domain model, annotation models address challenge `I` by relaxing the most troublesome restriction when passing from a domain model to a set of annotation types: relations between annotations. Entities representing annotation types in an annotation model are associated to elements in the Java language AST; just like annotations are related to the program elements on which they are placed. This association can be qualified by means of a query expression that will represent the code element to which annotations are supposed to be placed on. This partially addresses challenge `II`, since by using these queries, the annotation framework developer can express the mapping between annotation types and the code elements of the applications that will use them.

Finally, the entities in the annotation types are used to generate their reification (challenge `IV`). With this reification of the annotation types, the framework developer can then interpret annotated applications without resorting to the representation of annotations

[3] for Annotation Validation
[4] for Modeling Annotations

on the AST. In addition to this, since ModelAn and the reified annotations are based on Spoon, the reified annotations are accessible from the Spoon annotation processors.

ModelAn and AVal provide facilities to overcome the four challenges raised by annotation framework development. Since both ModelAn and AVal process annotations in the program's source code, they can specify and model annotations regardless of their retention policy (whether the annotations are kept only in source code, byte-code or at runtime). Also, reified annotations can be useful both for compile-time interpretation (using Spoon) or at runtime; although this last property is not yet implemented.

1.3 How to read this document

This document is divided in four main parts: in the first part we motivate the need for support in the development of annotation frameworks by exploring related domains and previous proposals in chapter 2 and in chapter 3 we present annotation frameworks and their development in detail. In the second part, we describe in detail our proposal for the development and validation of annotations frameworks: in chapter 4 we deal with the problem of annotation constraint definition and checking, while in chapter 5 we propose the use of domain models to streamline the development of annotation frameworks. In the third part, (chapter 6), in order to validate our approach, we apply it to three annotation frameworks. Finally, in the last part, we present the perspectives opened up by our work, and conclude (chapter 7). Each of the chapters are introduced in detail below.

Chapter 2: State of the art In this chapter we start by comparing annotations as a development tool to three other domains: Domain-specific languages, model-driven engineering and aspect-oriented software development. We postulate that the annotation types that are defined as part of a framework in fact represent a domain-specific language. The domains of annotation framework engineering and domain-specific language development are similar both in methodology and implementation strategies. Nevertheless, annotation types lack a grammar that define their correct use. We also postulate that annotation types in fact represent a domain-model, and therefore, techniques and tools useful for model-driven engineering are also applicable to the development of annotation frameworks. However, annotation types count with numerous restrictions that impede their use as modeling entities. From both these views of annotations as domain-specific languages and annotations as models we base the main insights of this work: that annotation frameworks will benefit of grammar-like rules to validate their use, and that annotation frameworks will benefit from modeling concepts in their development and implementation. Finally, we compare annotations to aspects, since both serve to represent crosscutting concerns. But, in contrast to aspects, that deal with both scattering and tangling of concerns, annotations only provide un-tangling capabilities, since the annotations themselves are still distributed all through the code of the program.

Since we identify (in chapter 3.4) annotation validation as a challenge in framework development, we also discuss the field of program validation. We concentrate on the role of meta-data (or annotation) directed validations, since this is the approach we take in chapter 4 for the definition of annotation constraints.

We finally discuss the state of the art in annotation framework development, presenting three approaches: Modeling Turnpike, XIRC and ADC. We give an summary of their functions, and compare each one to our proposal.

Chapter 3: Annotation Framework Development In this chapter we introduce annotations and their development. We start by defining an annotation framework as a framework that uses annotations as principal means of interaction with the application that uses it. We identify two actors when dealing with annotation frameworks: on one hand the annotation frameworks developer, who is in charge of defining the annotation types and their interpretation, while on the other hand, there is the application developer, in charge of mapping the annotations offered by the framework to the abstraction in his application. To illustrate the development of annotation frameworks we introduce a small framework called SaxSpoon. SaxSpoon offers a set annotations for parsing XML documents. Using SaxSpoon we point out a number of challenges in the design and development of annotation frameworks. These challenges are addressed in chapters 4 and 5. Challenges identified are (`I`) The representation of domain concepts as annotation types, (`II`) the mapping of annotation types to code elements, (`III`) the definition of constraints to validate annotated programs, and (`IV`) the reification of annotations for their interpretation.

Chapter 4: Annotation Constraints In this chapter we address challenge `III`; namely the definition and checking of constraints in annotated programs. To be able to define annotation constraints, we first divide them in two groups: those constraints that annotations impose on other annotations (*annotation-wise* constraints), and constraints that annotations put on the code elements on which they are placed (*code-wise* constraints). Having done this, we define a set of generic annotations for both groups. The generic annotations represent constraints commonly found on the specification of annotation frameworks; for example, that an annotation can only be placed on fields of a certain type, or that the use of an annotation in a class prohibits that class from carrying another annotation.

In order to provide the annotation framework developers with an implementation of these generic constraints, we introduce the AVal annotation framework. The idea is that a constraint for an annotation is in fact a meta-datum, and as such, it can be represented as a meta-annotation[5]. AVal offers a set of annotations, one for each generic constraint, that serve as domain-specific language for specifying annotation frameworks. AVal also implements an annotation processor that checks the constraints on an application's source-code; whenever an annotated element violates a constraint, AVal will report the error as if it was a compilation error. Finally, AVal is integrated into the Eclipse IDE.

We use the SaxSpoon annotation framework defined in chapter 3.2 to illustrate the application of the generic constraints and AVal annotations to the specification of the framework.

[5] An annotation on an annotation's definition

Chapter 5: Modeling Annotations In this chapter we address the remaining challenges, `I`, `II` and `IV`; namely, representation of domain concepts as annotation types, mapping of annotations to code elements, and reification of annotations. We do this by introducing the concept of annotation models. An annotation model is defined by making the relationships between annotations explicit. By doing this, annotations achieve a higher level of abstraction, bridging the gap between concepts of the domain and the annotations that represent them in applications. Annotation models are realized by extending the definition of annotation types through a meta-annotation called `Association`. Since annotation models are derived from the annotation types of the framework, the use of these annotation types on a given application defines an instance of the annotation model. By means of the `Association` meta-annotation, an interpretation engine, called *ModelAn* is able to construct the annotation model for a given set of annotation types. It also produces a source-code processor that takes an annotated application and produces the corresponding annotation model instance.

Annotation models allow the framework developer to state the default values of elements of the annotation types (`Default` meta-annotation), they also permit to define the relationship between annotations and elements of the code (`Targets` meta-annotation), and finally serve as reification for the application's interpretation. We also exploit the tools for model constraint validation to complement the generic constraints defined in chapter 4. The use of ModelAn is illustrated by means of the SaxSpoon annotation framework.

Chapter 6: Case Studies In order to validate the use of AVal and ModelAn, we apply them to three industrial and research annotation frameworks: Fraclet, Java WebServices, and the Java Persistence API. Fraclet defines six annotations for the development of primitive components in the Fractal component framework. Java WebServices (JWS) is a Java specification[6] for the implementation of web services; it defines seven annotation types. Finally, the Java Persistence API (JPA) is a part of the EJB3 specification that deals with the persistence of entities. To do so, 64 annotation types are included in the specification. Of these, we analyze ten.

From these three case studies, using a combination of AVal constraints and ModelAn meta-annotations, we defined each annotation model, and for each one, we specified the constraints derived from the specification (in the case of JWS and JPA) or from the developers of the annotation framework (in the case of Fraclet).

Chapter 7: Conclusion and Perspectives Finally, in this chapter, we present possible avenues for future work, such as an expansion on the number of generic constraints by analyzing other annotation frameworks, and advanced model extraction by interpreting annotation types in a different manner.

[6] JSR 181

Part I

State of the Art and Motivation

2

State of the Art

Contents

As discussed in the introduction (chapter 1), annotation frameworks act as extensions to the Java language, by providing high-level domain-specific abstractions embedded in the source code of an application. In this chapter we will start by discussing other approaches to achieving these kind of abstractions, namely works on the area of Domain-Specific Languages (DSL), Model-Driven Engineering (MDE) and Aspect-Oriented Software Development (AOSD). Having discussed the context in which annotation frameworks lay, we present several works on annotation framework development which are directly comparable to the one presented in this thesis.

Software development strives to enhance programmer productivity. This is a consequence of the cost reduction of computations, where raw efficiency gives way to quicker

development of quality, maintainable applications. In this regard, ways to bridge the semantic gap that exists between the concepts of the problem domain, and those that are manipulated by the programmer are sought both by academia and industry. Several approaches to this problem are the subject of research; domain-specific languages, model driven development, aspect orientation and annotation frameworks are just some of them.

***

In this chapter, we postulate that Domain-Specific Languages, Model-Driven Engineering and Aspect-Oriented Software Development are in fact closely related to annotations, and that the tools and techniques developed for each of these areas can be of benefit when developing annotation frameworks. We discuss each of these areas in sections 2.1, 2.3, 2.4 and 2.6. In section 2.5 we also review related work in the field of program validation, of which annotation validation lays, and in section 2.2 we discuss several program transformation tools and techniques that can be used for the development of annotation frameworks; in particular, Spoon, which is used to implement the tools proposed in this document.

2.1 Domain Specific Languages

A domain-specific language (or *DSL*) has been defined as "a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain." [vDKV00]. In this definition, the goal of the DSL is stated to be that of providing "focused expressive power", through which a DSL programmer is supposed to bridge the abstraction gap mentioned earlier. Domain-specific languages have a number of characteristics that give advantages on their use over general-purpose languages (*GPLs*). Because they are focused on a particular domain, their notions and abstractions are better suited to solve problems in that domain than general abstractions. Applications that use DSLs can also be safer, in the sense that since the language provides tailored abstractions for a domain, it can also provide tailored checks that validate domain specific constraints.

When defining what a DSL is, it is necessary to contrast it with general purpose languages; however, the distinction between DSLs and GPLs can indeed be a fuzzy one. Although it can be argued that languages such as COBOL and FORTRAN are DSLs for the scientific and business domains, they are normally not considered as such. DSLs tend to be small languages (they are also called mini-languages [Ray03] or little languages [Ben86]) and they tend to be *declarative*; of these two, COBOL and FORTRAN are neither. Nevertheless, the nature of a DSL is also in the eye of the beholder; some DSLs can be used to implement applications outside of their intended domain, thereby losing their domain-specificness.

2.1.1 Domain-Specific Language Development

Regardless of where on the domain-specific vs. general purpose spectrum a language lays, the development of DSLs remains largely an ad-hoc endeavor. Methodologies on how to develop DSLs have been proposed in literature: Lengauer et al. [LBCO04] proposed a structured way to develop DSLs based on the notion of program family [Par76]. A program family is a set of programs that share enough characteristics that they can be taken as representative of a domain. The idea is then to, based on a given program family, identify concepts, assumptions and computations common to the family (and therefore to the domain represented by the family). Then, from the program family, a library is refined, and from the library an abstract machine that defines domain-specific instructions and domain-specific state is defined. The final step is then to guide the design of the DSL through the abstract machine and the program family: common programming patterns suggest syntactic constructs, the abstract machine gives insights to how to interpret programs, and the implementation of the library can give insights on possible optimizations.

A more general approach to the development of DSLs is proposed by Mernik et al. in [MHS05] where they identify four phases in the development of DSLs, and for each a set of patterns is proposed. In the *Decision* phase, the choice of constructing a new language is made. Patterns to aid in this decision include whether existing DSLs can fulfill the requirements, and give common reasons to opt for a DSL, such as task automation or GUI construction. The second phase, *Analysis* deals with identifying the domain for the new language by relying on domain experts or domain analysis techniques such as the program families discussed previously. Patterns for this phase include formal and informal domain analysis. The third phase is the *Design* of the language; which can rely on existing languages or not (language exploitation and language invention patterns) and whether the language is formally or informally described. Finally, the *Implementation* phase proposes as patterns interpretation, compiler/application generators, a preprocessor, embedding the language in a host (general purpose) language or hybrid approaches.

2.1.2 Relationship with annotation framework development

Going back to the definition of a DSL presented at the beginning of this section, it is not difficult to see the relationship between annotation frameworks and DSLs. DSLs provide abstractions focused on a domain, while annotations also provide abstractions generally linked to a given domain in the form of extensions to the semantics of the code elements of the Java language. Also, annotations (as DSLs) tend to be small and declarative (when a programmer annotates a piece of code, he only states what it represents, and not how to interpret it). Even the implementation strategies for the interpretation of annotated programs have parallels with those described in section 2.1.1; annotations can be interpreted at compile-time with a preprocessor, or at runtime by an actual interpreter. Because of this, one can think of annotation frameworks as embedded DSLs that extend the semantics of the Java language. Despite this, calling annotation frameworks DSLs might be a stretch, given that they do not explicitly define a language, since no grammar is defined for it. If each of the annotations in a framework is likened to a term in a DSL,

no link between them can be made.

Having introduced the field of Domain-Specific Languages and their development, and discussed its relation to annotation frameworks, in the next section we explore the relation between annotations and models in the context of Model-Driven Engineering.

2.2 Program Transformation

Several of the implementation patterns for DSLs presented in [MHS05] deal with program transformation: be it to implement the DSL by translating it to operations on a base language, using a macro preprocessor, or extending an existing compiler; program transformation tools are needed. As previsously discussed, the interpretation of annotations in a program is similar to the implementation of DSLs therefore, in this section we present some of the better known program transformation tools in the light of annotation framework development.

There exist several well-established program transformation frameworks, we will concentrate on those dealing with term rewriting, and on those that are based on compile-time reflection. In the term-rewriting camp, some of the most representative are TXL [Cor06b] and ASF-SDF [vDHK96], both these frameworks are based on term rewriting and include some form of concrete abstract syntax (or *native patterns* as they are called in TXL). TXL offers several transformation architectural styles [Cor06a], these architectures permit either a sequential application of transformations (*cascade*), or a parallel one (*aspect*). In ASF-SDF, the notion of traversal functions [vdBKV01] is included to control the way in which the rewrite rules are be applied. They allow for bottom-up and top-down. Traversal functions can also be distinguished by whether they changed the tree they travel (*trafo*), extract information from it (*accu*) or both. Both TXL and ASF-SDF can be used to implement the interpretation of annotation framework. However, they do not provide any specific facilities to do so.

Stratego/XT [BKVV06], while being similar to ASF-SDF, provides a sophisticated set of additional features, such as concrete syntax templates, rewriting strategies and dynamic rules (for non context-free rewrite rules). In particular, the Dryad library [KBV08] complements Stratego with a bytecode interface that allows the translation of compiled Java classes to Stratego's term grammar, and from this grammar to bytecode. It also includes an experimental type-checking front that annotates Java expressions with its compile-time type. It is important to note that these type-annotations are not related to annotations in the Java sense; they are attributes of the terms representing the expressions in the AST. Typing information is important when developing interpretation engines for annotation framework development, since it is common for annotations to refer to classes (types) in the program, and to be constrained by them. The use of attributed AST terms for program transformation is the base of another rule-based tool called JastAdd [EH07]. JastAdd however (as of this time) does not fully support Java5, and therefore its use as an annotation framework interpretation engine is limited.

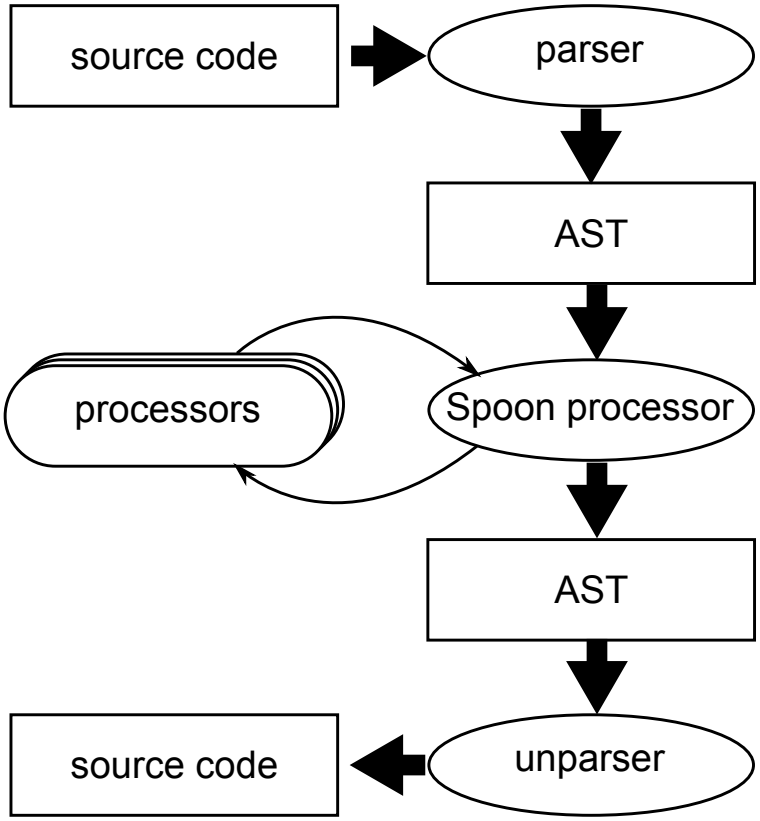


Figure 2.1: Spoon process flow

2.2.1 Spoon

Compile-time reflection tools are based on the programmatic manipulation of the AST of the program as reflection of runtime entities. While several tools exist [TCKI00, Chi95, LH01], we will concentrate on Spoon [PNP06, Paw05][7], since it will be used to implement the approaches proposed in chapters 4 and 5. Spoon is a compile-time reflection based program transformation tool, geared towards the processing and interpretation of annotated programs.

Spoon [PNP06, Paw05] is a source-code processor based on a meta-model of the program that models every code element, including statements and expressions. It relies heavily on generics to ensure type safe processing, and uses the concept of *processors* as units of program analysis and transformation. The process flow of a Spoon run is shown in Figure 2.1. The source code of a program is passed thorugh Spoon's parser, which extends the Eclipse JDT parser. From this, an AST is obtained. The AST is then visited by a number of processors that will perform the actual program transformation. Each processor is passed once, and at the end of the processing round, a final Spoon processor will pretty-print the AST back to valid Java code.

[7] http://spoon.gforge.inria.fr/

In each visiting, the processor has complete (both read and write) access to the model. Special processors are `AnnotationProcessors` that declare the annotation in which they are interested, and the type of elements on which the annotation is applied. A processor that goes over methods annotated by a `Test` annotation is shown in figure 2.2.

```
public class TestProcessor extends
     AnnotationProcessor<Test,CtMethod>{

  public void process(Test annotation, CtMethod meth){
    // Testing code
  }

}
```

Figure 2.2: Spoon annotation processor for the `Test` annotation on methods

Since Spoon's model is tied to the Java AST, it cannot be used to analyze programs written in other languages as term-rewriting based frameworks. In addition to this, since Spoon relies on a Java compiler, it requires the analyzed application to be compilable; that is, in addition to require the source code to be syntactically correct, it requires the libraries that are used by the application. Nevertheless, the use of JDT as first step provides Spoon with a robust name and type resolution implementation that is updated with each evolution of the Java language. In addition to this, Spoon provides advanced services such as Concrete-Syntax templates for code generation and querying, and a static analysis engine that provides a precise intra-procedural flow-graph.

Spoon is used in a number of projects. SpoonGraffiti[8] [FN07a] uses annotations to implement call-back futures in distributed applications, AOKell[9] [SPDC06] is an implementation of the Fractal component framework that uses Spoon to optimize the meta-programming layer of the components, and JDiet[10] eases the development of J2ME applications by transforming their source code.

An extension of Spoon, programmed by Barais [Bar06], models Spoon's Java meta-model in the Eclipse's Modeling Framework. This extension, called SpoonEMF, takes input files, and from them produces an EMF model which can be processed by Spoon processors in the same manner as with normal Spoon models. By doing this, the SpoonEMF developer can manipulate the source code of a program via the Spoon API, the EMF API or other EMF-aware langauges such as Kermeta [DFFV06]. For example, in [MJCH08] Kermeta is used to implement a measurement system for models, and it is applied on SpoonEMF-born models.

[8] http://spoon.gforge.inria.fr/SpoonGraffiti/Main
[9] http://fractal.objectweb.org/tutorials/aokell/
[10] http://spoon.gforge.inria.fr/JDiet/Main

2.3 Model-Driven Engineering

A model is defined as "a simplified representation of an aspect of the world for a specific purpose" [Jéz08], In terms of software engineering, models are used as a mechanism to cope with the complexity of large applications. The use of models in programming is not new, going back to CASE tools in the 1980's. They are used as a way to communicate by providing a common language among participants of the development of an application; as a specification tool where properties of the final application are defined in terms of its model; and as an implementation aid, either serving as blue-prints for the application, or by transformation it to a (possibly partial) application. Since models abstract representations, models that abstract from other models are used. Such models of the structure models are called meta-models , and they are pivotal in the model world [FEBF06]. Domain models [SIP+05, RM06] are used to represent the concepts of the domain of a given application, and in this sense, they are similar to DSLs. Indeed, programs written in a DSL can be likened to domain models, where the DSL itself corresponds to the domain-model's meta-model.

2.3.1 Model Development

Model-Driven Engineering *MDE*, consists of the placement of models at the heart of software development. The idea is that models are useful for all the stages of software development, from high-level requirements engineering, design, implementation, testing, deployment, and maintenance and evolution. In MDA [MM03], the Object Management Group defines a specification of an architecture for MDE, based on the Unified Modeling Language (*UML*) [Obj04]. In it, the development of applications starts by the definition of a *computation independent model* (CIM), which is a domain model that represents the environment of the application and its requirements. Complementing it, a *platform independent model* (PIM) which captures the structure and behavior of the application without committing to a particular technological platform is constructed. Using the PIM, a model that incorporates technical details of a particular platform is constructed (PSM). This final model can then be translated into executable code in the target platform. Although the MDA specification singles out three models, there might be intermediate models between them whenever this is deemed convenient by the model architect, for example, detailing non-functional concerns which then are weaved into the other models to obtain a complete representation of the application [Jéz08].

One of the attractive points of using a model-oriented approach to software development is the ability of defining consistency constraints in terms of the models manipulated at different abstraction levels. One way to express these constraints is through the use of the Object Constraint Language (OCL) [Obj06]. OCL allows the developer to express constraints that represent invariants on the state of modeled elements, or queries over them. In [CGQ+06] Costal et al. define a number of patterns or generic constraints that make the expression of common constraints in OCL easier to specify. These common constraints are similar to the ones we define in this thesis for the purpose of validating annotation frameworks in section 7.1.2.

2.3.2 Relationship with annotation framework development

The relationship between annotations and models (and MDE) can be seen in two axes. First, annotations can be likened to domain models. As mentioned before (section 2.1.2), annotations define a set of domain-specific abstractions just as domain models do. It can be argued that the annotation types defined in the framework are to their use in an application, as a domain meta-model is to a domain model. That being said, domain models remain at higher abstraction level than annotations, and they are not held back by the restrictions that the Java language imposes on annotation types (no inheritance, no associations between annotation types, etc.)

The second axis is how annotations fit into the model-oriented development methodology, that is, how to include annotations in the generation of applications. Annotations, seen as meta-data attached to a code entity, are semantically close to stereotypes as defined in UML 2.0 [Obj04]. Indeed, it is common to represent annotations, during design, as stereotypes [CK05]. Nevertheless, it is difficult to establish a direct mapping between stereotypes and annotations given the particularities of annotations. For example, annotations do not allow for inheritance, an annotation can be placed on different code elements (stereotypes are restricted to one[11]), and most importantly, annotations can refer to types that are defined in the program to 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.

In this section we have postulated that annotation frameworks are based on a domain model, and that the development of annotation frameworks would benefit from tools and techniques existing in the domain of Model-Driven Engineering. In the next section we present how annotations are related to the field of Aspect-Oriented Software Development, since both provide support for the modularization of crosscutting concerns.

2.4 Aspect-Oriented Software Development

Annotation frameworks can also be compared with Aspects in the Aspect-Oriented Software Development domain. Aspects [BCC05] modularize crosscutting concerns by factorizing them into an *advice* which is then weaved into a set of places (*joinpoints*) in an application as defined by a query expression (*pointcut*). Annotations can be seen as a kind of aspect, the advice being the computations performed when interpreting the annotated application, and the pointcut the set of places where the annotations are placed in the application. Aspects provide two main properties: untangling and un-scattering [KLM+97]. Untangling means that the concerns present in the application are well separated from each other, while un-scattering means that concerns are locally in the same place, and not distributed all-over the application. Of these two properties, annotations provide

[11] This is true for stereotypes as defined in UML 1, in newer versions of the specification (UML2), this restriction is relaxed.

untangling, but not un-scattering; since the concerns that the annotations implement are separated from the code of the application (and put into the annotations), but the annotations themselves remain distributed all over the application. The relationship between annotations and aspects is further discussed in [FN07b].

Aspects can also be used to interpret annotations. In this configuration, the application programmer explicitly enumerates the places on which an advice is to be applied by annotating methods. An aspect will then use a pointcut that groups all occurrences of an annotation and insert the corresponding advice. This use of annotations to declare crosscutting concerns is detailed in the work by Kiczales and Mezini [KM05].

In the next section we will discuss the topic of program validation, since we have found that the validation of the constraints of annotations is an important step on the development of annotation frameworks.

2.5 Program Validation

The validation of the use of annotations in a program is an important step on the development of annotation frameworks. Since annotations can be seen as extensions of the Java language, non-specialized Java compilers[12] perform limited tests on their use, leaving them to the interpretation engine of the framework. In this regard, annotation framework development borrows from the program validation domain. In this section we will explore the use of annotations in the validation of programs.

Static validators allow developers to check properties of their code that go beyond of that what is provided by normal compilers. Lint [Joh78] 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 [EGHT94], and later Splint [EL02], guide the validation of programs through annotations (stylized code comments) that explicit programmer assumptions and intents.

In [Hed97], 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. More recently, Eichberg et al. [EKKM08] propose a mechanism to check structural constraints in programs using a set of facts in a Datalog database. Structural properties are defined using annotations and gathered into overlapping ensembles which are then checked by Datalog queries.

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 [SY02]. Nevertheless, in an Aspect Oriented language such as AspectJ[HH04], no extra reflection facilities are provided, so the validation programmer must rely only on Java reflection which does not reify the body of methods.

In the next section we will explore the state of the art in annotation framework development. Approaches in this domain are rooted either on the use of models, or on the

[12] Like Sun's javac `http://java.sun.com/javase/technologies/core/toolsapis/javac/` or Eclipse's JDT `http://www.eclipse.org/jdt/`

application of program validation principles.

2.6 Annotation Framework Development

Annotation frameworks is the name that we use for frameworks that provide interaction with the application through annotations. Such frameworks are a kind of what is known as *active libraries*. Also known as semantically enhanced libraries, or library-level optimizations, *Active Libraries* [VG98] 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 and high-performance computing [Vel98, FJ97], and several *generic* active library definition frameworks, Broadway [GL04] and Pivot [Str05] have been proposed. These frameworks provide tools and abstractions so that the library programmer can make its library *active*.

As discussed in previous sections (2.1.2, 2.3.2 and 2.4), annotation frameworks share a number of similarities with Domain-Specific Languages, models in Model-Driven Engineering and aspects in Aspect-Oriented Software development. Annotations in an application can be seen as a kind of embedded DSL, which is then interpreted using techniques similar to those employed for DSL implementation. This being said, a grammar that links the different annotations in an application cannot be defined, since no provisions for it exist in the Java language. Annotations can also be seen as a modeling tool, where different concepts from a problem domain are mapped to elements of the Java language. Annotations, however, are not expressive enough to be effectively used as a modeling language: individual annotations can only define data and not behavior, and no relation between annotations can be defined. Finally, annotations can be seen as markers that represent the places on an application in which special computations must take place. In this case, annotations are more similar to aspects, with the interpretation of each set of annotations being the advice, and the annotated program elements, the pointcuts. When compared with aspects, annotations provide a degree of untangling of the code, since the non-functional behavior is kept separate from the code of the annotated element. But, unlike aspects, they do not provide unscattering of the non-functional behavior, since the annotations themselves are still present all over the application. In addition to this, the use of annotations breaks the concept of obliviousness since the application programmer is aware that an "aspect" in the form of an annotation will apply at a given place in the code. Whether this lack of obliviousness is a drawback of annotations with respect to aspects is a matter of discussion.

Tools and techniques for the development of annotation frameworks are scarce in academic literature. In industrial applications, the only support for their development is that given by the Java SDK in the form of the Annotation Processing Tool and the Pluggable Annotation Processing API. Both these tools give means to process annotations present in an application, but give no aid in the development of the annotation types themselves; leaving to the annotation framework developer the task of the design, specification, validation and interpretation. In this section three approaches to the development

of annotation frameworks are presented: Modeling Turnpike, XIRC and Attribute Dependency Checker. They are compared in terms of whether they provide higher-level abstractions for annotations and annotation types, whether they allow the declaration and checking of constraints, and whether they give support for the interpretation of the annotations.

2.6.1 Modeling Turnpike

The use of models for the development of annotation-based programs is explored in [WS05] by Wada et al. They propose a full MDA approach that starts from a model, and ends with an executable program. Modeling Turnpike (or mTurnpike) is divided into two parts: the front-end system that maps domain-specific concepts from a modeling layer to a programming layer; and the back-end which uses model and program transformation to go from the programming layer to actual executable programs. In terms of annotation frameworks, the front-end of mTurnpike describes an annotated program, and the back-end its interpretation by transforming it to a un-annotated program.

mTurnpike's front-end is based on the concepts of *Domain-Specific Model (DSM)* and *Domain-Specific Code (DSC)*. A set of UML2 stereotypes represent the annotation types provided by the framework, which in mTurnpike, is called a DSL. The DSM is a UML Class diagram that uses the stereotypes defined in the DSL. From the DSM a set of Java classes that compose the DSC are generated by mapping UML classes to Java classes, stereotypes to annotations, etc. The skeleton classes of the DSC are then edited by a programmer, filling out the code for the methods. Transformations allow to go back to the DSM from a given DSC.

The back-end takes the DSM and DSC and generates a set of Java classes that can be finally run. The UML class diagram that contains the DSM is transformed to one in which the stereotypes in the classes are "unfolded" into associated classes that do not carry the stereotypes (by for example translating a «Remote» stereotype in a class into a super interface `java.rmi.Remote`). This unfolded UML diagram is transformed into Java code. Since the UML diagram does not contain behavioral information, the generated methods are left empty. The generated Java classes are then combined with the DSC, in order to fill out the body of the methods. When the combination is finished, the result should be a complete Java program.

Discussion The mTurnpike system gives a complete framework for the development of annotated applications. It provides high-level constructions (the DSM) as well as low level ones (the DSC) so that core concerns are separated from the domain-specific ones. It also allows the specification of the transformations through which the annotations in the program are interpreted.

The system, however, supposes that the annotation types (the DSL) already exist, and therefore provides no aid in their development. The idea of modeling annotation types as stereotypes, while natural, might not be the most appropriated one (as discussed in [CK05], where a class representation is preferred). In addition to this, the fact that the annotation-types are explicitly modeled is not exploited fully; no definition of consistency constraints is made, nor is checking of the validity of annotations in the DSC

performed. Several implementation issues in the relationship between the DSM and DSC are not clear: in the DSL (stereotype model) relationships between stereotypes are allowed; however when the stereotypes are used in the class diagram, no mention of these relationships is made. To better explain the problem with the representation of relationships between annotations, suppose that mTurnpike is used to develop an annotation framework to develop component-oriented applications. Two stereotypes are defined in the DSL: `Component` and `Interface`, and a `provides` relation between them. Now, suppose that an application developer uses this framework by annotating DSM composed of a `Client` class with `Component`, and two other classes `Naming` and `Location`. There is no information on to which class is `Client` related by the `provides` relation. Now, the information on the relation can be very important when transforming the DSM and DSC to actual executable code.

2.6.2 XIRC

In [ESM05], Eichberg et al. outline a mechanism through which structural properties (constraints) of classes can be checked. Their approach, called XIRC, is based on an XML tree representation of an application's code for which XPath queries that represent the properties to be checked are run. If the XPath query returns a non-empty set, then the resulting entities are flagged as violating the structural property. Using XIRC, the authors define and check structural properties of the annotations defined in the EJB3 Java specification [MK06].

XIRC works by first creating the XML representation of the application from its byte-code, this is delegated to the Magellan [EMOS04] framework. Then the XPath queries that define the constraints are run. For convenience, XIRC allows "context defining" queries that factorize commonly used expressions to reduce repetition on the definition of each constraint. The XIRC framework is integrated into the Eclipse IDE compilation process, and the constraint violations are presented in a transparent manner to the application developer.

Discussion The use of XPath as a constraint definition language in XIRC seems motivated by the application representation format, XML. A more declarative language, such as OCL, could be more suited for this task. Regarding the use of XIRC to check annotation constraints, several questions arise: first, the authors only check constraints imposed by the use of annotations on code elements (for example that a class being annotated @Entity cannot be final) and do not give examples of constraints between annotations themselves (for example that an annotation cannot be placed on an element that already sports another annotation). While nothing suggests that this kind of inter-annotation constraint checking is not possible using XIRC, the lack of a specialized annotation representation (such as mTurnpike's DSL model) might make the definition of annotation constraints difficult.

In addition to this, the decision of extracting the code model from the compiled byte-code of the application limits the scope of annotation frameworks that can be checked with XIRC. Annotations in Java can be made to exist only in source code, byte-code or runtime; extracting them from the byte-code leaves out those annotations that are meant

to be source-code only. As it stands, the only place on which all annotations can be found remains the source code of the application. Related to this, one of the possible targets for annotation are local variables. Since the local variable information is lost when the application is compiled to byte-code, XIRC will not be able to perform checks on the validity of its use.

Finally, the definition of XIRC's constraints on a separate file might go against the philosophy of annotation-based development. Indeed, annotations provide a way to unify programing artifacts by, for example embedding external configuration files on the source code of the application. By using XIRC, the advantages of this use of annotations are nuanced since the developer finds himself with an additional external file to manage when trying precisely to reduce the number of such files.

2.6.3 Attribute Dependency Checker

Microsoft's .Net platform [Pro02] provides a programming construct similar to annotations in Java called custom attributes [LX07], Cepa et al. introduce in [CM04] the Attribute Dependency Checker (*ADC*) tool to validate dependency constraints between .Net's custom attributes. The ADC is based on the idea of meta-attributes to declaratively define the dependency constraints between attributes. It defines a single meta-attribute called `DependencyAttribute` which, when placed on an attribute definition, allows the developer to state which other attributes are required or disallowed at different levels (assembly, class or method). The `DependencyAttribute` does not define *how* these dependencies are checked, since it leaves the implementation to other tools, in this case the ADC. ADC is a *post-processor* that uses the .Net reflection API to interpret the `DependencyAttribute`s and raise errors when the specified dependencies are not met.

Discussion The constraints offered by `DependencyAttribute` are limited to the relations already present in the AST of the program: assembly-class, class-method. No support is given for other kinds of inter-attribute relations (such as the ones between attributes in two different classes). In addition to this, no extensions to the checking mechanism are provided, so it is not possible to define more complex checks (for example requiring one and only one attribute on a classes' method).

Although it is outside of the scope of ADC, a large number of constraints for annotation/attribute frameworks deal with the relationship between the annotations/attributes and the code on which they are placed. This makes the use of `DependencyAttribute`s of limited utility to specify a framework.

2.6.4 Comparison of each of the approaches

Having introduced the existing approaches to aid in the development of annotation framework, we will now compare them to the one we propose. The comparison will be made in three main axes: the representation of the annotated program, that is, what are the abstractions to manipulate the program and the annotations in it, and whether all annotations are supported; the constraints that are defined and checked, that is, whether constraints are declarative or explicit, and whether annotation-wise or code-wise constraints

are supported; and what kind of support for the interpretation of annotated programs is provided.

The comparisons between mTurnpike (section 2.6.1), XIRC (section 2.6.2), and ADC (section 2.6.3) are summarized in tables 2.1, 2.2 and 2.3. Each table is followed by a description of the criteria used.

	Platform	**Representation**		
		Code	**Annotation**	**Support**
mTurnpike	Java	model	model	yes
XIRC	Java	XML	none	no[a]
ADC	.NET	DOM	none	yes

[a]Only represents annotations present in byte-code

Table 2.1: Annotated application representation in the compared approaches

Platform whether the approach is based on the Java or .Net platform.

Code Representation the way in which the application is represented; AST, a model, an XML document.

Annotation representation whether annotations are represented explicitly in an special manner other than their code representation.

Annotation Support whether the approach supports all possible uses of the annotations.

	Constraints			
	Code	**Annotation**	**Declarative/ Explicit**	**Embedded/ External**
mTurnpike	no	no	–	external
XIRC	yes	no[a]	explicit	external
ADC	no	yes	declarative	embedded

[a]No explicit support for annotation-wise constraints

Table 2.2: Constraints offered by the compared approaches

Code Constraints whether the approach provides support for the constraints that annotations impose on the code elements on which they are placed.

Annotation Constraints whether the approach provides support for the constraints that annotations impose on other annotations.

Declarative Constraints whether the constraint definition is made in a declarative or explicit way.

	Interpretation		
	Support	**Compile-time**	**Runtime**
mTurnpike	yes	transformations	–
XIRC	no	–	–
ADC	no	–	–

Table 2.3: Interpretation support offered by the compared approaches

External/Embedded Constraints whether the constraint definition is embedded in the source code of the annotation framework or if it is defined externally.

Interpretation Support whether the approach provides support for the interpretation of annotations.

Compile-time interpretation support for the interpretation of annotations at compile-time.

Runtime Support support for the interpretation of annotations at runtime.

Of the analyzed approaches, all but one (ADC) are targeted to the Java platform. All of them provide different representations of the code (models, XML files, the CodeDOM reflection API of .Net), but only mTurnpike provides explicit representation for the annotations in the application. mTurnpike represents annotations as stereotyped UML class diagrams. The utility of stereotypes to represent annotations is disputed in [CK05], where class-based approach is preferred.

None of the approaches allow for the definition of constraints both between annotations and between annotations and the code on which they are placed. XIRC does not prohibit constraints that deal with annotations only, but it doesn't provide special facilities to do it. Of the three approaches that permit the definition of constraints, only none accommodates for both declarative and explicit definitions, leaving the developer the choice to use whichever is better fitted for the task at hand.

In terms of interpretation support, only mTurnpike provides it. Compile-time interpretation support in mTurnpike is provided by the transformation of the stereotyped model to Java skeleton classes and then the merging of the resulting code with the domain-specific code

Finally, none of the approaches provide any support for the runtime interpretation of annotated programs.

2.7 Summary

In this chapter we have presented different approaches that help developers to cope with the complexity of systems; in particular domain-specific languages, and models and model-driven engineering. We have discussed the relations between them and annotation frameworks, pointing out their differences and similarities (sections 2.1 and 2.3). Annotations

provide a sort of domain-specific language that extends the general purpose one; in this case Java. However, for annotations to be seen as a language, their grammar must first be defined; the tools for which are not provided by the Java language. We then compare annotations to models in the model-driven engineering sense. While annotations provide abstractions that stem from a domain model, they are at a lower abstraction level than that of models. In addition to this, annotations again suffer from the restricted language in which they are expressed; indeed, associations between annotations and ways to define their semantics (and consistency constraints) are needed. The task of defining and validating consistency constraints source code is covered by the program validation domain, which is discussed in section 2.5, where we look at how annotations are used to support program validation, and present other ways to validate programs.

Having defined the context on which annotation framework development lays, we explore different approaches to their implementation in section 2.6. In it, we present three proposals: Modeling Turnpike (section 2.6.1), XIRC (section 2.6.2) and the Attribute Dependency Checker (section 2.6.3). They are compared approach in section 2.6.4, where we argue that none of them provide support for the all the different phases of annotation framework development, Modeling turnpike being the one that arrives the closest to that goal.

In the next chapter, the task of annotation framework development is discussed in detail. The actors in the development of annotation frameworks and their interests are identified, the composition of an annotation framework is discussed, and by means of an example, a number of challenges in the design, implementation and interpretation of annotations are enumerated.

3

Annotation Framework Development

Contents

As stated in chapter 1, annotation frameworks are composed of two main parts: the set of annotation types, and their corresponding interpretation engine. The annotation types serve as the interface (or API) of the annotation framework, they are the entities that are manipulated by the application programmer. The interpretation engine can be likened to the implementation of the API defined by the annotation types. In this regard, annotation frameworks provide a stronger separation between the interface of the framework and its implementation than traditional frameworks that rely on Java interfaces and extension relationships. In traditional frameworks the interaction between

the application and the framework is made explicitly by means of method calls to the API, or extension of framework classes; while in annotation frameworks this interaction is made implicitly and declaratively through the use of annotated elements in the application's code. Indeed, annotation frameworks and traditional ones are not exclusive in the sense that it is possible (and common even) to use both kinds of mechanisms, allowing the framework developer to use whichever interaction mechanism is better suited for each case. An example of this is the Hibernate framework [BK06].

From the application developer's point of view, the use of an annotation framework consists of deciding which annotations (provided by the framework) should be placed on which elements of the application. This decision requires a good understanding of the concepts that each annotation type represents from the application developer. This is needed so that the application developer can correctly map the concepts of its application (represented by classes, packages, fields, etc.) to those provided by the annotation framework; making a parallel to traditional frameworks, if a developer wants to use a framework, he has to first study the concepts of the framework to know what methods to call, or which classes to extend. The grasp of the semantics of the annotation framework is also important because the way in which annotations are placed on the application must comply with the assumptions of the annotation framework; this, in turn, is similar to the requirement often present in traditional frameworks that require certain protocols to be respected when invoking services of the framework (for example calling an `open()` operation on a stream before being able to `read()` from it).

***

In this chapter we will concentrate on the development of pure annotation frameworks, that is, those that exclusively rely on annotations. The process described in this chapter can be extended to frameworks that use other kinds of interaction in addition to annotations. The chapter is organized as follows: first we discuss the two main parts of an annotation framework (section 3.1), namely the annotation types and their interpretation. Then we show the complete process of developing an annotation framework by introducing an annotation framework for the development of SAX parsers in section 3.2 and discussing two possible interpretation engines, one at compilation time and one runtime in section 3.3. Finally we discuss a number of challenges raised by the development of annotation frameworks in section 3.4.

3.1 Anatomy of an Annotation framework

Just as with any framework, when confronted with the task of developing an annotation framework, the developer must concern himself with two tasks: the interface, and its implementation. In annotation frameworks, annotation types are the interface, and annotation interpretation its implementation. Both components are discussed in this section, but we first discuss the actors that deal with annotation framework development.

3.1.1 Actors

In the development of annotation frameworks it is important to distinguish two actors: the developer of the framework, called the *annotation framework developer* and the developer who uses the annotation framework, called the *application developer*. This distinction is important since it is easy to confuse the two types of development. Obviously, it is not required for these two actors to be single different individuals; but rather roles that developers (or teams of) can take. In order to clarify the remainder of this chapter, we outline the tasks and interests of each of these actors.

Annotation Framework Developer This actor is responsible for the design and implementation of the annotation framework. The annotation framework developer must be fluent in the domain for which the framework provides services, so that it can correctly represent their concepts as annotations; as well as in metaprogramming techniques required for the interpretation of annotated programs. In addition to this, the annotation framework developer needs to produce the means for the application developers to correctly use his framework; this comprises the documentation of the framework, and development environment support.

Application Developer This developer is the user of the annotation framework, and as such requires knowledge in the way in which the annotation framework expects to be used. That is, the constraints and semantics of each annotation. He also needs to define the way in which the concepts of his application relate to the concepts represented by the annotation framework. For this, the application developer relies on the documentation of the frameworks, as well as on the support given by his development environment.

3.1.2 Annotation types

Annotation types in Java are defined in a similar way to Java interfaces. An annotation type exists within a package, has a name (qualified by that of its containing package) and it contains a number of elements. For a complete description of annotation types, please refer to the Java Language specification [GJSB05].

The code shown in Figure 3.1 contains the definition of an annotation type (lines `1` through `9`) and its use on an application (lines `13` and `16`). As mentioned before, annotation types are similar to Java interfaces, as can be seen on its definition in line `3`. This particular annotation, called `MyAnnotation`, defines three elements, `annotation`, `value`, and `options`. The annotation type also defines an inner `enum` called `Options` (line 8).

Annotation type elements named `value` carry special semantics in Java; normally, when an annotation is used, the element/value pairs must be defined as `element = name`. If it is the case that the only element that carries a value is the one called `value`, then the name of the element can be omitted, as is the case in line `13`. Also, it is possible to omit elements that have default values when using the annotation type, as is the case with the elements `annotation` and `options` in line `13`, and `annotation` in line `16`.

```
1 package myPackage;

3 public @interface MyAnnotation {
4   AnotherAnnotation[] annotation() default {};
5   String value();
6   Options options() default Options.O1;

8   enum Options {O1, O2}
9 }

11 //...

13 @MyAnnotation("aValue")
14 public class AClass{

16  @MyAnnotation(value = "aValue", options = MyAnnotation.Options.O2)
17  int foo;

19 }
```

Figure 3.1: An annotation type definition and use

3.1.3 Restrictions and limitations of Annotations

There are a number of restrictions imposed by the Java language to the definition and use of annotations. First of all, annotation types cannot inherit from other annotation types, nor can they implement interfaces. This restriction impacts the expressive power of annotations, as generalization/specialization of concepts represented by annotations is impossible. Annotation type's elements are also restricted: the types allowed for elements are limited to those that can be expressed at compile time; namely primitive types (integers, strings, etc), classes, other annotations and arrays of those types. In the case of elements which contain annotations, due to the fact that inheritance is prohibited for annotation types, elements that contain annotations must contain a single annotation type; i.e., it is not possible to have an annotation type element to contain several different annotations. This makes it impossible to define the annotation type equivalent of a generic collection. As for the annotation use, there is a limited number of code elements which can carry annotations. These are: Packages, types (including annotation types and enums), constructors, methods, fields, method and constructor's parameters, and local variables. For each of these code elements, a single annotation of each type is allowed.

3.1.4 Annotation interpretation

Having defined the interface for the annotation framework, in the form of a set of annotation types, the semantics of the framework must be defined. In traditional frameworks, the semantics are implemented by a set of classes that provide the services offered by the framework's interface. In annotation frameworks, semantics are implemented in an interpretation engine. This engine takes as input an application whose elements (classes, methods, etc) carry annotations, and performs the operations directed by them. In prac-

tice, interpretation engines take care of several concerns: firstly, they take care of validating that the use of the annotations in the application respects the constraints defined by the framework; this is done either explicitly, by checking the constraints and reporting violations back to the programmer, or implicitly, by having the interpretation fail (sometimes silently). Apart from validating the input, annotation frameworks sometimes build an in-memory representation of the annotations present on the code. This allows the annotation framework developer to distance himself from the code of the application, and to add additional information which is not present on the annotations (such as relationships between the different annotations). Finally, the annotation framework performs the actual interpretation which can happen either at compile or runtime. Compile-time interpretation is performed by generating additional code, or modifying that of the annotated application. At runtime, the annotation framework can also use the information present on the annotated application at runtime to modify the way in which it responds to service requests. Examples of both kinds of interpretations are given in sections 3.3.1 and 3.3.2.

In both compile-time and runtime interpretations, annotation frameworks must recur to some kind of metaprogramming facility. In the case of runtime interpretation, the metaprogramming API used is normally Java's reflection API. Annotations by default are not reproduced in the classes' byte code, in order to change this, the annotation framework developer must annotate its annotation types with `@Retention(RetentionPolicy.RUNTIME)`. As for compile-time interpretation, the annotation framework developer resorts to source code preprocessors in order to find and manipulate the annotations; The Annotation Processing Tool (APT) and Spoon are examples of annotation processors that provide compile-time reflection and metaprogramming. Both runtime and compile-time interpretation are discussed next.

Compile-time interpretation

Compile-time interpretation of annotated programs is performed by an external tool in the compilation chain. Normally it is done as a pre-compilation step in a manner similar to configuration file processing in traditional frameworks. The annotations in a compile-time interpretation are used to direct the transformation or generation of code additional to that of the annotated application. The generated code implements the functionality declared by the annotation types. For example, if the annotation framework's purpose is to enable the persistence of objects in an application, then the annotations will specify the mapping between the classes of the application and the schema of the database. In this case, the compile-time interpretation will generate the necessary code to connect to a database, and commit, update or query it in order to persist or retrieve the objects represented within.

Several tools to enable compile-time *processing* of annotated programs exist. These are called *annotation processors*. With the release of Java 5, Sun made available the *Annotation Processing Tool* (APT). This tool allows the processing of annotations using the reflective facilities of Java. The tool supports only the generation of code, and not its transformation, since it does not provide access to the full AST of the program, not does it allow its modification. The Spoon [Paw05] remedies this by introducing full compile-time

reflection [TCKI00] and annotation processing facilities.

Runtime interpretation

Annotations by default are kept in the compiled bytecode, but are not retained by the virtual machine at runtime. To change this, Java's API offers an annotation called `Retention` which instructs the Java compiler to reproduce code annotations in the byte code, and to make them available at runtime through the reflection API. Therefore, runtime interpretation is only possible for annotation types that carry the `Retention(RetentionPolicy.RUNTIME)` annotation. Other possible values for the `Retention` annotation are `SOURCE`, which instructs the compiler not to reproduce the annotations in the bytecode, and `CLASS` which is the default behavior.

Annotations are accessible at runtime by the `getAnnotation()` method defined by the `java.lang.reflect.AnnotatedElement` implemented by the runtime reflections of annotatable elements. This method receives an annotation type as parameter, and returns a dynamic proxy with the values of the annotation if present, and `null` otherwise. This means that runtime annotation processing is restricted to code elements which have a reflection object, namely packages, classes, constructors, methods, their parameters, and fields. Local variables, although annotatable, are not accessible through runtime reflection, so their interpretation at runtime is impossible.

It is important to note that, while the interpretation of the annotation in the application is carried out at runtime, the annotations remain compile-time entities. This means it is not possible to assign new annotations to code elements nor change their value at runtime. In addition to this, it is important to remember that annotations cannot be placed on objects, but on the classes that define them, that is, two objects of the same class will carry the same annotations. These two restrictions limit the utility of runtime interpretation of annotated programs, since the dynamic adaptation of annotations cannot be performed.

One could imagine a mechanism, similar to the current annotations in Java, that allows the attachment of meta-data to runtime entities. This mechanism would overcome the restrictions that limit the usefulness of runtime interpretation of annotations by allowing the modification and late-binding of meta-data at runtime. These *runtime annotations* would compose a meta-object facility that enhances the expressiveness of the language. Such runtime annotations however, would fall outside of the scope of this work, since we concentrate on the development of annotation frameworks as they are currently defined in the Java language. Concepts, as defined in [Der05], are defined as elements of the knowledge of a domain mapped to code elements are similar to the hypothetical runtime annotations described here, and could provide a base for their implementation in the Java language.

In the following section, we illustrate the development of an annotation framework by means of an example. This example will define the set of annotation types that composes

the annotation framework's interface, and present two possible interpretation engines, one at compilation time using Spoon, and one at runtime using reflection.

3.2 SaxSpoon - an Annotation Framework for XML Parsing

To better explain the nature of annotation framework development, we present a simple annotation framework, called SaxSpoon. SaxSpoon is a compile-time annotation framework that aids the programmer in the construction of XML manipulation classes that use the *Simple API for XML (SAX)* [MB02] 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 SAX programmer is interested in the start of several tags, he must place the tag handling code for each tag on the same `startElement()` method, which reduces its cohesion since the method will manage the handling of different tags. In this case, a normal approach is to separate the code for each tag into a private handler method, and have the `startElement()` method be a large `case` statement that dispatches to the correct handler method in function of the name of the tag being handled. The construction of the `startElement()` method is then repetitive, and therefore error prone.

For the `characters()` and `endElement()` methods, a similar problem arises. Since the `endElement()` method is called for each closing tag, the method has a similar form as that of the `startElement()` method, with the same drawbacks, and a similar solution pattern. In the case of the `characters()` method, the SAX specification states that the parser is not forced to up-call the `characters()` method with the full content of tag, but that it can split the contents of the tag into several chunks that result in multiple up-calls. Because of this, the full contents of the tags is only known when the `endElement()` method is invoked. To address this, SAX developers usually accumulate the data that is given through the `characters()` method, and process it in the `endElement()` method.

SaxSpoon aims to rid the SAX developer of the repetitive, error prone tasks of writing the `startElement()`, `characters()`, and `endElement()` methods, and concentrate on individual methods for the handling of each individual tag. The main idea is then for the SaxSpoon developer to write two methods for each tag (one for the start of the tag, and one for the end of the tag), and use annotations to instruct SaxSpoon which methods handle which tags. SaxSpoon then invokes the correct methods whenever the start or end of a tag are found. The start tag handler methods, have one parameter per attribute of the tag that they handle, while the end tag handlers have a single parameter that represents the content (characters) contained on the tag. More than contribute to the state of the art of XML technology, SaxSpoon's goal is to provide a concrete example of a simple, yet not trivial annotation framework.

The annotations that are provided by SaxSpoon, as well as two interpretation engines (one compile-time and one runtime) will be discussed next.

Annotation	Target	Elements	Description
`XMLParser`	Class implementing `ContentHandler`	*dtd*	Marks a class as a SaxSpoon 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 3.1: Overview of SaxSpoon annotations

3.2.1 SaxSpoon annotation types

Table 6.2 shows the three annotations defined by SaxSpoon. `XMLParser` marks a class implementing the `ContentHandler` interface as a SaxSpoon class. It takes as a parameter the DTD file that describes what kinds of documents the class can handle, this is used to automatically validate the incoming XML document. The `HandlesStartTag` and `HandlesEndTag` annotations mark methods belonging to classes annotated with `XMLHandler` as either start tag handlers or end tag handlers.

3.2.2 SaxSpoon example

To illustrate the use of SaxSpoon to process XML documents we show the following example. Suppose a class of XML documents that define cooking recipes. In figure 3.2 the DTD that defines valid XML recipes and an example recipe are shown.

In the code listing in figure 3.3, a `TestParser` class is defined in which SaxSpoon annotations are used to pretty print a recipe. In line `1`, a `XMLParser` annotation defines the `TestParser` class as using SaxSpoon. As a parameter, the annotation states which DTD will be used to validate recipes. The `TestParser` class extends `DefaultHandler`, which indirectly implements the `ContentHandler` interface, both are provided by Java's implementation of SAX. In lines `4` through `12` two methods are defined, `startRecipe` and `endRecipe`, that will handle the start and end events for the `recipe` tag. They are marked by the SaxSpoon annotations `HandlesStartTag` and `HandlesEndTag` respectively. Each of the `Handles` annotations has as parameter the name of the tag which they handle.

Notice that the `startRecipe` method in line `5` defines no parameters, since the `recipe` tag as defined in the DTD in listing 3.2 defines no attributes. In contrast, the `startNutrition` method in line `16` defines a number of parameter congruent with the attributes for the nutrition tag stated in the recipe DTD.

The use of SaxSpoon to the define the XML handler brings a number of advantages over simple SAX use. First, each of the methods handles a unique tag, which enhances their cohesion, and understandability. Second, thanks to the `HandlesStartTag` and `HandlesEndTag` annotations, the purpose of each of the methods is explicitly stated, just by looking at the source code it is possible to know which method handles which tag. Finally, the repetitive, error prone writing of the `startElement`, `characters` and `endElement` methods is relegated to

recipe.dtd

```
<!ELEMENT collection (description,recipe*)>

<!ELEMENT description ANY>

<!ELEMENT recipe (title,
           ingredient*,
           preparation,
           comment?,
           nutrition)>

<!ELEMENT title (#PCDATA)>

<!ELEMENT ingredient
       (ingredient*,preparation)?>
<!ATTLIST ingredient
       name CDATA #REQUIRED
       amount CDATA #IMPLIED
       unit CDATA #IMPLIED>

<!ELEMENT preparation (step*)>

<!ELEMENT step (#PCDATA)>

<!ELEMENT comment (#PCDATA)>

<!ELEMENT nutrition EMPTY>
<!ATTLIST nutrition protein CDATA #REQUIRED
              carbohydrates CDATA #REQUIRED
              fat CDATA #REQUIRED
              calories CDATA #REQUIRED
              alcohol CDATA #IMPLIED>
```

margarita.xml

```
<recipe>
  <title>Margarita Cocktail</title>
  <ingredient name="tequila"
              amount="1.5" unit="Oz"/>
  <ingredient name="triple sec (Cointreau)"
              amount="0.5" unit="Oz"/>
  <ingredient name="lime juice"
              amount="0.5" unit="Oz"/>
  <preparation>
    <step>
     Rum the rim of a cocktail glass
     with lime juice, and dip in salt.
   </step>
   <step>
     Shake all ingredients with ice,
     strain into the glass, and serve.
   </step>
  </preparation>
  <nutrition
    calories="153"
    carbohydrates="7g"
    fat="0"
    protein="0.2"
    alcohol="0.25"/>
</recipe>
```

Figure 3.2: Recipes DTD and example

```
1 @XMLParser(dtd = "http://localhost/recipe.dtd")
2 public class TestParser extends DefaultHandler {

4  @HandlesStartTag("recipe")
5  public void startRecipe() {
6    System.out.println("Recipe");
7  }

9  @HandlesEndTag("recipe")
10 public void endRecipe(String chars) {
11   System.out.println("--");
12 }
13 //...

15 @HandlesStartTag("nutrition")
16 public void startNutrition(String calories, String protein, String fat,
17                 String carbohydrates) {
18   System.out.println("Nutrition Facts");
19   System.out.println("\tProteins\t " + protein);
20   //...
21 }
22 }
```

Figure 3.3: SaxSpoon class to process Recipes

the framework's interpretation engine, which will be discussed in the next sections.

3.3 SaxSpoon Interpretation

To illustrate the interpretation of annotation frameworks we present two implementations for the annotation types defined in SaxSpoon: one takes a compile-time program transformation approach, and the other a runtime approach using Java's reflection facilities. The fact that two completely different interpretations are possible without changing the annotation framework's interface is a testament to the decoupling between annotation types and their interpretation, which is one of the advantages of using annotation frameworks.

Both interpretation techniques offer different advantages and drawbacks. On one hand, the compile-time approach, relying on code generation, produces an application which is more efficient than the runtime one, since it does not use costly runtime reflection. On the other hand, the runtime approach is easier to debug, since no hidden code is generated which can be foreign to the application's developer. Both approaches, nevertheless, require a validation of the annotated program to assure that the interpretation will succeed. In these examples no special constraint validation will be performed to keep the example clear; however, the places in which the violation of constraints will make the interpretation fail will be pointed out. We start off by describing the compile-time implementation of the interpretation engine.

3.3.1 Compile-time interpretation of SaxSpoon applications

The first interpretation engine is one based on the transformation of classes that use SaxSpoon into classes which use purely the SAX API. To this end, the Spoon program transformation framework is used. The basic idea behind this program transformation is to generate the `startElement`, `characters` and `endElement` methods, as depicted in figure 3.4. The generation of each of the SAX API methods is discussed next.

startElement() This method dispatches each of the incoming start tag events to the corresponding method as stated by the SaxSpoon `HandlesStartTag` annotations. The body of the `startElement` method is then a set of `ifs` that identifies the method that handles the incoming tag. Once the method identified, each of the attributes of the tag is retrieved and given as a parameter to the handler method. If the handler method does not define the correct names and number of parameters (i.e., those defined in the DTD of the XML document) the interpretation engine will generate incorrect (possible uncompilable) code.

characters() In this method, a buffer is used to accumulate the characters present between tags. The buffer will then be passed as a parameter to the `endElement` method.

endElement() For this method, a dispatch technique similar to that of the `startElement` method is used. A set of `ifs` that invoke the correct handler method (as defined by the `HandlesEndTag` annotations) for each tag. The current characters buffer is sent as a parameter to the handler method. As with the `startElement` method, if the

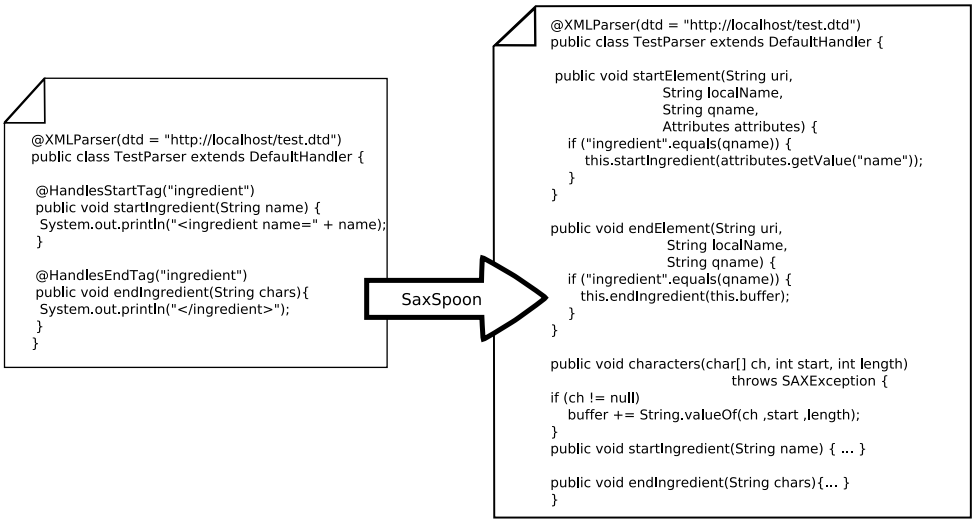


Figure 3.4: Code transformation for SaxSpoon programs

handler method does not define the correct parameter, the interpreter will generate incorrect code.

The code transformation/generation is realized using Spoon by means of one annotation processor, and six source code templates; ensemble, they total almost two hundred lines of code.

3.3.2 Runtime interpretation of SaxSpoon applications

In order to show how an annotation framework can be interpreted in a runtime environment, we present an alternative to the compile-time interpretation presented in the previous section. The idea behind the runtime interpretation of SaxSpoon classes is the same as for the compile-time one; namely dispatching events to their method handlers as directed by the `HandlesStartTag` and `HandlesEndTag`.

In this interpretation engine, a single class, `ReflectiveSaxDispatcher` wraps an instance of a class annotated with `XMLParser`, and uses reflection to implement the `startElement`, `endElement` and `characters` methods. The source code of this class is discussed below.

```
1 public class ReflectiveSaxDispatcher extends DefaultHandler{

3 Object wrapee;
4 StringBuffer charactersBuffer;

6 public ReflectiveSaxDispatcher(Object wrapee) {
7 //...
8 }

10 public void startElement(String uri, String localName, String name,
11                 Attributes attributes) throws SAXException {

13 Method[] methods = wrapee.getClass().getMethods();
```

```
15 for (int i = 0; i < methods.length; i++) {
16  Method method = methods[i];
17  HandlesStartTag hstt = method.getAnnotation(HandlesStartTag.class);
18  if(hstt != null){
19    if(hstt.value().equals(name)){
20        //reflectively invoke method
21        }
22   }

24 }
25}

27public void characters(char[] ch, int start, int length)
28        throws SAXException {
29  charactersBuffer.append(ch);
30}

32public void endElement(String uri, String localName, String name)
33        throws SAXException {
34 super.endElement(uri, localName, name);

36 Method[] methods = wrapee.getClass().getMethods();

38 for (int i = 0; i < methods.length; i++) {
39  Method method = methods[i];
40  HandlesEndTag het = method.getAnnotation(HandlesEndTag.class);
41  if(het != null){
42    if(het.value().equals(name)){
43      //reflectively invoke method
44        }
45    }
46 charactersBuffer = new StringBuffer();
47}

49}
```

The wrapped object, and the buffer in which the contents of each tag will be stored are defined as fields in lines 3 and 4. The `startElement` method in line 10 uses Java's reflection to traverse the methods of the wrapped class (lines 15- 24) and for each one, tests if the method carries a `HandlesStartTag` annotation (lines 18- 19) and if that is the case, it invokes the method, using as parameters strings containing the attributes of the tag. As with the compile-time interpretation presented before, if the handler method does not define the correct number or order of parameters, the interpretation of the annotation will fail. The implementation of the `characters` method is identical to the one generated by the compile-time approach. Finally, the `endElement` method's body is similar to the `startElement`'s; the methods of the wrapped class are traversed, looking for the one that handles the current tag. Once it is found, it is invoked with the contents of the buffer as parameter.

Compared to the compile-time interpretation, the runtime-one is more succinct, totaling under fifty lines of code. However, given its heavy use of reflection it is considerably

slower . Notice that both interpretation techniques require the validation of the annotated class in order to correctly behave; this suggests that the validation of annotated programs *is independent of the technique used* to interpret them.

3.4 Challenges in annotation framework development

Taking SaxSpoon as example, we can illustrate a number of challenges that exist in the development of annotation frameworks. These challenges involve both the annotation framework developer and the application developer, and they are rooted in the design and implementation of the frameworks.

3.4.1 Design

The task of designing an annotation framework is, in essence, a mapping from the concepts of the domain that the framework is supposed to represent to annotation types, and to the annotatable elements in a program. In the case of SaxSpoon, the concepts of the framework are XML parsers, start tag handlers and end tag handlers. Each of the concepts is mapped to an annotation type. In addition to this, each annotation type must be mapped to an annotatable code element; in the case of SaxSpoon, XML parsers are mapped to classes, while start and end tag handlers are mapped to methods.

The mapping of concepts from the framework to annotation types presents the first challenge to address when designing an annotation framework. While it is true that this task is similar to the task of mapping concepts of a domain to classes; something which is well understood and commonly used in development, when mapping concepts to annotation types, the annotation framework developer must deal with the fact that annotation types are less expressive entities than classes. Indeed, restrictions such as lack of inheritance in annotation types, restricted types for the elements of annotations, and the fact that annotation types only carry data and not behavior, make the mapping to annotation types harder than the mapping of concepts to classes.

The second challenge in designing annotation frameworks comes from the mapping of the annotation types to code entities. The annotation framework developer must decide which of the code elements (packages, classes, methods, etc) is more suited to represent the concept embodied by an annotation type. When considering this, the annotation framework developer must regard the ensemble of annotation types defined, since usually the relationships existing between annotation types will be realized by the relationships between code elements. For example, in SaxSpoon, tag handlers belong to a XMLParser; this relationship is realized in the program by the fact that the `XMLParser` annotation is mapped to classes, the `Handles*Tag` annotations are mapped to methods, and classes contain methods; ergo, `XMLParsers` contain `Handlers`.

A third challenge we have identified deals with the validation of annotated applications. When the framework developer writes the annotation types and decides their mapping to code elements, a number of implied rules on the use of the annotation by the application developer must be respected. To illustrate this, take the `HandlesStartTag` annotation in SaxSpoon. This annotation is mapped to methods, but in addition to this,

for the annotation to be meaningful, the method must belong to a class which carries the `XMLParser` annotation. The constraints in the use of the annotation types are defined during design, since they regard either the annotation types, that is the valid values for their elements, or the code elements on which they can be placed in an application. The definition of these constraints is important, since their violation impedes the correct interpretation of an annotated program. The annotation framework developer needs then a way in which he can define constraints for his annotation framework, and a way in which to check them prior to the interpretation phase.

3.4.2 Implementation

Challenges in the implementation phase of the development of an annotation framework vary depending on the particular framework. Different annotation frameworks require different interpretation techniques. Hybrid frameworks, i.e., those that use annotations as well as other means to interact with the framework, will normally require a runtime implementation. Frameworks in which annotations are used more as configuration artifacts favor compile-time interpretations; while other annotation frameworks such as SaxSpoon, can use either. In all cases, a powerful metaprogramming facility is desired, since all annotation frameworks require to reason about the program that they interpret. In this regard, an interesting challenge arises when considering how annotations present in a program are manipulated. By default annotation processing tools rely on a simple reification of the annotation which is a 1-to-1 mapping of the annotation type. This means that to interpret the `HandlesStartTag` annotation, the annotation processing frameworks offer an object that contains only the elements declared in the annotation type. This reification does not reflect the original design of the annotation framework, since associations between annotation types are lost; which, for the SaxSpoon example, makes it very difficult to know to which `XMLParser` does a handler method belongs.

Aside from this, annotation framework interpretation remains a very case-specific task, and identifying phase-wide challenges is difficult.

3.4.3 Challenges

In summary, the challenges identified are:

I The representation of domain concepts as annotation types

II The mapping of annotation types to code elements

III The definition of constraints to validate annotated programs

IV The reification of annotations for their interpretation.

3.4.4 Proposal

Taking into account the challenges raised above, we put forward the proposal of this thesis, which proposes to ease the development of annotation frameworks by providing tools and

techniques that will help the annotation framework developer in the design, specification, representation and validation of the framework.

This will be achieved in two steps: first by studying existing annotation frameworks we propose a set of generic constraints common to annotation frameworks. These constraints are realized through the development of an annotation framework, called AVal, whose annotations are used to specify the annotation types of an annotation framework. AVal then interprets the annotated program, reporting violations to the constraints to the developer. This step in the approach is described in chapter 4.

The second step consists on borrowing tools and techniques from the MDE field to define a model of the annotation types belonging to an annotation framework. This *annotation model* is then used to specify the mapping of annotations to code elements and to reify annotations in an application. Both goals are realized by the ModelAn annotation framework. This step is further described in chapter 5.

Our proposal provides solutions to all the challenges identified: AVal covers challenge `III` while annotation models and ModelAn cover challenges `I, II` and `IV`.

3.5 Summary

In this chapter we have given an overview of the way in which annotation frameworks are defined by analyzing the components of an annotation framework and discussing the different strategies for the interpretation of annotated applications. We have provided an example of an annotation framework to illustrate its development process, and from this process we have identified a number of challenges in both the design and implementation of annotation frameworks. Challenges identified are (`I`) The representation of domain concepts as annotation types, (`II`) the mapping of annotation types to code elements, (`III`) the definition of constraints to validate annotated programs, and (`IV`) the reification of annotations for their interpretation. Based on this four challenges, we put forward a proposal that addresses this issues.

In the following chapter we address challenge `III`, i.e., the definition of annotation constraints. We will discuss the nature of annotation constraints, and propose an annotation framework for the definition and evaluation of these constraints on annotated programs.

Part II

Proposal

4

Annotation Constraints

Contents

The Java type system for annotation is not expressive enough to ensure that the use of an annotation framework is correct. This type system allows the annotation framework developer to define the names, types and default values of possible properties, and 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, such as EJB3 [MK06], impose constraints on the use of annotations that go beyond the capabilities offered by the Java programming language. For example, 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`. Rules such as these are common among annotation framework specifications. These kinds of rules 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.

***

In this chapter we address the problem of validating annotated programs, identified in chapter 3.4.4 as challenge `III`. In the first part of the chapter, we start by defining the different kinds of constraints that an annotated program must comply with in section 4.2. Such constraints are specific to each annotation framework. In order to ease the task of constructing annotation frameworks, we have identified a number of *generic constraints* (section 7.1.2) which are common to annotation frameworks. In the second part, we propose a mechanism, based on annotations, for the definition of the constraints of annotation frameworks (section 4.5), and their validation in annotated programs section 4.6.

4.1 Validating annotation constraints

As discussed before, annotation frameworks imply a number of rules that govern the way in which they are used. Indeed, this is not different than as with 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, rule checking in annotation frameworks is considerably easier than that of regular frameworks because, in general, no complex static analysis must be performed.

We shall name the process of annotation constraint checking *validation of an annotated program*, and the process of checking a single constraint, a *validation*. This process, depicted in Figure 4.1, takes as inputs the set of annotation types and program carrying the corresponding annotations and the set of constraints, defined over the annotation types and checked against the program. As output, a set of errors corresponding to the violations of the constraints as they are used in the program. In this process we identify two actors: the developer of the annotation framework, i.e., the person that implements the annotation types; and the program developer, i.e., the person that writes and annotates the program. Since the annotation developer defines the semantics of the framework, it is up to him to also define the constraints that their use must comply with.

Although the process flow for the validation of an annotated program is straightforward, the actual performed validations vary greatly in function of the particular annotation framework. Indeed, each annotation framework counts with its particular set of constraints that derive from the domain in which they lay. In order to derive a generic approach to validation, commonalities among the different constraints defined by different annotation frameworks must be found. Firstly, we classify the constraints annotation frameworks in two kinds. Then, we propose a number of generic constraints in each one.

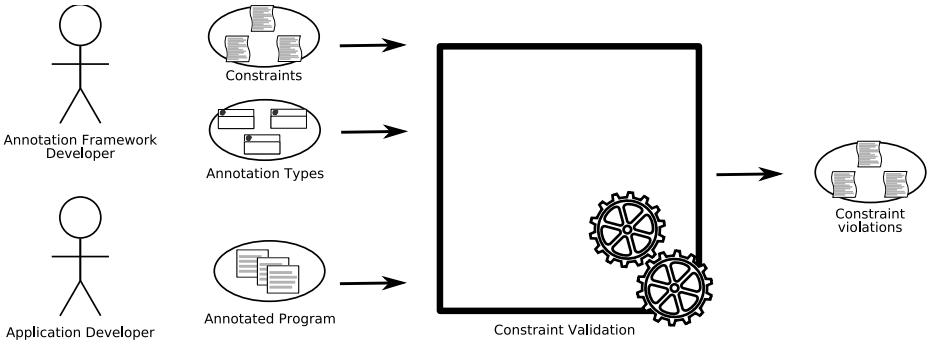


Figure 4.1: Validation process flow

Each generic constraint is parameterizable to the context of each annotation framework. By using these generic constraints, an annotation framework developer is able to specify the semantics of his framework.

4.2 Kinds of constraints

In the general case, 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 with the annotation type's properties, and its relationship with other annotation types. The former are named code-wise validations, while the later annotation-wise validations.

Annotation-wise Validations Structural rules define the relationship between annotations and between an annotation and its properties. In this case, a distinction is made between the former (relationships), and the later (value validations). Value validations restrict the possible values of the properties of the annotation type, for example, having an integer within a certain range, or a string conforming to a regular expression.

We identify two types of annotation-wise relationship constraints: those constrained by scope or by reference. We define the scope of an annotation as the AST nodes of the sub-tree of the element on which the annotation is placed (see Figure 4.2). This way an annotation on an element can have a relation with annotations placed on elements within its scope. In this sense, annotations placed on a method are within the scope of the annotation placed on the class to which the methods belong. To better illustrate the concept of scope, consider the relation rule between entities and ids in EJB3: an id must be *inside* an entity.

References express relationships between annotations which are on different scopes. They are normally specified through a special value on a property, either an identifier, for example the name of the referenced annotation, or a type which carries a given annotation. For example, in EJB3 relationships between entities are specified by, among others, the annotation `OneToOne` that takes as an attribute the type which itself must be an entity.

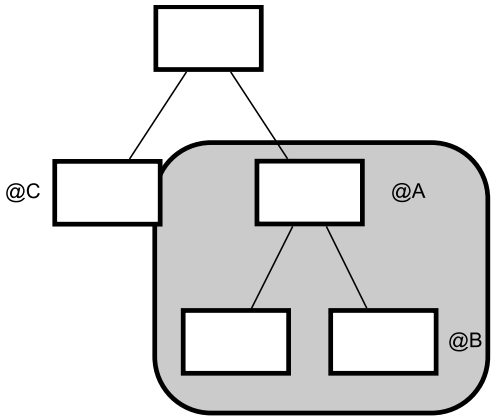


Figure 4.2: In this AST, annotations `B` is in the scope of `A` while `C` is not. The scope of `A` is represented by the gray square.

Code-wise Validations Code-wise validations deal either with the *target* of the annotation type, that is the kind of code element on which the annotation type is allowed, or with the characteristics of said target: the visibility of the class, the declared return type of the method, etc. Given that annotations are always placed on code elements, code-wise rules of usage are always present, albeit with varying degrees of complexity.

In general, code-wise validations are orthogonal between each other, i.e., the code-wise rules for each annotation can be checked independently from the others. They, however, depend on the code in which they are, requiring at times non-trivial source code analysis.

As an example of a code-wise validation, let us consider a rule that states that a certain annotation `@A` can only be placed on fields which are collections. In this case there are two different rules: one stating that the target is a code element of the kind field, and the second one is a restriction on the type of this field. The first rule can be validated in a straightforward manner, but a complete validation of the second one, may require a complex analysis to derive the runtime type of the field in the case in which its static type is not a direct subtype of a collection.

4.3 Generic constraints

Even though the set of constraints that define the validity of each annotation framework varies in function of each framework's particular needs, we have encountered that similarities between them arise. We therefore propose a number of *generic constraints* that can be reused and adapted to represent the individual needs of each annotation framework. It is worth noting that this set of generic constraints does not cover all the possible constraints, and that additional constraints remain a necessity. Generic constraints we identified are summarized in Table 4.1, and explained below. A formalization on the semantics of these constraints is given in appendix A.

Generic constraints reason about three kinds of entities: annotation types, their instances which are placed on AST nodes, and the types defined and used on the program that is to be checked. Each of the generic constraints identified are described in the

Annotation-wise
`Inside`
`Prohibits`
`Requires`
`Unique`
`RefersTo`
Code-wise
`Target`
`Type`

Table 4.1: Generic constraints

following sections.

4.3.1 Annotation-wise generic constraints

Annotation-wise constraints restrict the placement of annotations relative to other annotations (`Inside`, `Prohibits`, `Requires`, etc.) or the valid values of their elements. In both cases, the validation of annotation-wise constraints is linked to their position on the AST and to the concept of scope introduced in the previous section.

Inside As explained previously, one annotation being inside of another one, is a commonly expressed constraint in annotation frameworks. An `Inside` constraint takes as parameters two annotation types, and requires that each instance of the second one to be placed within the scope of at least one instance of the first one. To clarify this constraint, suppose that two annotation types `A` and `B`, the first one placed on classes, and the second one on methods; so that every method annotated with `B` must belong to a class annotated with `A`. In this case, we would state the constraint `B` *inside* `A`.

Prohibits The placement of an annotation in a node can prevent other types of annotations from being placed on that same node. This is common when annotation types represent exclusive properties or concepts. If it is the case that an annotation type `A` and an annotation type `B` cannot be placed on the same AST node, then `A` *prohibits* `B`.

Notice that the *prohibits* relation is symetric: although it is `A` that prohibits `B`, this is equivalent to stating that `B` *prohibits* `A`. This constraint specifies that nowhere in the program, a node can have both `A` and `B` annotations.

Requires Similar to the prohibits predicate, the placement of an annotation on a node can require another one. This is expressed as `A` *requires* `B`, and its application checks that whenever a node is annotated with an instance of `A`, that node must also carry an instance of `B`.

In contrast with the prohibits generic constraint, *requires* is not symmetric: `A` *requires* `B` does not imply that `B` *requires* `A`. In the first case, it is correct to find a node annotated with `B` and not `A`, while in the second one, the same case would be a violation of the constraint.

RefersTo The three generic constraints introduced above rely on the relations of the AST tree to specify the constraints by using either the scope (inside) relation or the target relation (prohibits and requires). Nevertheless, it is often the case that more complex constraints require relations not present in the AST, when this is the case, a reference is used. We define references as functions that map an annotation of one type to a set of annotations of (possibly) a different type. The corresponding constraint is that the cardinality of the resulting set is to be greater or equal to one.

Referencing annotations can be done in two ways: by naming annotation instances, using the value of an element as identifier, or by using a type.

Id references create links between annotation instances by matching a value of an annotation's element with the value of another one (possibly of a different type). The associated generic constraint takes four parameters: a starting annotation type, a target annotation type, and the respective elements that link them. This way, if elements `eA` and `eB` of annotation types `A` and `B` respectively serve as link between them, then instances of `A` and `B` that have the same value are linked. This is represented as the generic constraint `A.eA` *refersTo* `B.eB`, and it is valid when for all instances of `A`, there exists at least one corresponding instance of `B`.

Type references are variations of the id references in which the value of the element that binds the annotations is a Java type that carries an instance of the referenced annotation. In this case, the link takes three parameters: the annotation type, the element defined in it, and the annotation type that goes on the Java type. Therefore, if the annotation type `A` has an element e whose value must be a Java type that carries an annotation of type `B`, then we say that `Ae` refersTo `B`.

Unique If it is the case that the id reference represents a one-to-one relation, then it is desirable to express a constraint that requires all the values of an element in an annotation type to be unique. This is achieved through the unique generic constraint. This constraint takes as parameters an annotation type and an element that is defined in it. If the values of an element `e` in an annotation type `A` are required to be unique, we say that `A` *unique* `e`.

4.3.2 Code-wise generic constraints

Code-wise constraints deal with the relation between the annotation and the program on which it is placed. The generic constraints we identified are those that restrict the AST node on which the annotation is placed (`Target`) and those that restrict the type of the node (`Type`).

In contrast with annotation-wise constraints, code wise validations can be harder to check because they can reason about the behavior of code at runtime. Take for example a possible constraint that states that methods that carry a certain annotation cannot create threads. The checking of this constraint statically requires non trivial analysis, since it is necessary to find all possible execution paths originating in the annotated method, and to assure that in none of them a thread is spawn. In addition to being complex, constraints that deal with the runtime behavior of annotated code elements, such as this one, are eminently specific to the annotation library in question, therefore difficulting its

generalization. Because of this, we have decided to delegate such checks to specialized static analysis engines, and concentrate on the generic constraints enunciated above, and explained below.

Target Target is the simplest kind of code-wise constraint. It constraints the kind of AST node on which annotations can be placed, to for example `Class` or `Field`. This generic constraint is actually provided as a part of the Java JDK[GJSB05]. Note that it is possible to constraint annotation types to more than one target, or even to constraint annotation to be used purely as elements of other annotations *i.e.*, no target.

Type Type related constraints can be of several kinds depending on the target of the annotation type. Type generic constraints have different semantics depending on whether the annotation is placed on a Type (class or interface), field, method, parameter or local variable.

- **Types** if an annotation type A targeted to classes or interfaces is constrained to a type T, then the type of the target must be a subtype of T.
- **Constructors** if an annotation type A targeted to a constructor is constrained to a type T, then T must be a super type of the class that the constructor instantiates.
- **Methods** if an annotation type A targeted to methods is constrained to a type T then the declared return type of the method must be a subtype of T.
- **Fields, parameters and local variables** if an annotation type A targeted to any of these is constrained to a type T, then T must be a super type of the declared type of the variable, field or method parameter.
- **Package** annotations that are placed on packages cannot be constrained by type.

4.4 Composition of Generic Constraints

Evidently, each annotation type belonging to a framework will require several instantiations of generic constraints to specify its semantics. Each of the generic constraints presented before is orthogonal[13], so the most common composition of the constraints is their conjunction, for example stating that an annotation `A requires B and A inside C`. It is also possible to express alternative constraints, via a disjunction or negations, like stating that `A requires B or A requires C`.

In addition to this, it can be of interest to the annotation framework developer to restrict the scope of one of the constraints by, for example, stating that an annotation must be unique in the scope of another one. The combinations are then reduced to the use of `inside` as a scope restriction. We define this scope restriction/extension only for the annotation-wise constraints; although it would be possible to express them also

[13]Each constraint is defined independently from the others. While this does not mean that they cannot contradict themselves, it allows for their independent checking

for code-wise constraints, we do not believe that such uses are meaningful as *generic* constraints.

Inside prohibits If an annotation A prohibits another one B inside the scope of a third one C, this means that not only it is forbidden to put A and B in the same AST node, but on any node which lies in the scope of C.

Inside requires In a similar way as the previous combination, if an annotation A requires another one B inside the scope of a third one C, then the scope of the requires constraint is extended to the scope of the inside of annotation C.

Inside unique Finally, it is possible to restrict the scope of the unique constraint in a similar way as the two above. If an annotation A must have a unique value inside B, then there cannot be other instances of A with the same element value under the scope of B.

Other kinds of scope restrictions could be imagined. For example, restricting the scope of the constraints to those nodes under a given package. We, however, have not found these kinds of scoping restrictions in the annotation frameworks analyzed, and therefore do not include them as generic constraints.

4.4.1 Example

To illustrate the use of generic constraints in an annotation framework, we get back to the SaxSpoon annotation framework defined in chapter 3.2. The constraints for each of the three annotation types in SaxSpoon, in natural language, are summarized in Table 4.2. In this table, the constraints for annotation types (below the name of the annotation type) and the constraints for each of the elements of the type are separated for clarity.

As shown on Table 4.2, the constraints to which programs using SaxSpoon must adhere to are divided into generic ones (annotation on class, type of method, etc) and specific ones (tags must be defined in a DTD). For each of the annotations, we will use the constraints defined in the previous sections to partially define the annotation's semantics. The specification of each of the annotation types in SaxSpoon is shown in Figure 4.3.

As we can see, most of the constraints of SaxSpoon can be specified using the generic constraints defined above. By doing this, the semantics of each of the annotations is clear to the application developer. In the next section (4.5) we will introduce an application called *AVal* that, based on the generic constraints described, allows the annotation framework developer to declaratively describe the semantics of an annotation framework by using a constraint annotation framework; and using this same framework, the application developer can check whether his annotated program complies with the annotation framework's constraints.

`XMLParser`

- Should be placed on Classes
- The class must (indirectly) implement the `ContentHandler`

dtd

- The string must be a valid URL.

`HandlesStartTag`

- Should be placed on methods
- The method should be of return type `void`
- The method's parameters should be all of type `String`
- The names of the methods parameters must be the same as the attributes of the tag handled by the method
- There can only be one Start method handler per tag
- A method cannot handle the start and end of a tag
- There should be a method that handles the end of every start tag

tagName

- The tag name must be a tag defined in the DTD of the `XMLParser` annotation

`HandlesEndTag`

- Should be placed on methods
- The return type of the method should be `void`
- The method should have a single parameter of type `String`
- There can only be one end method handler per tag
- A method cannot handle the start and end of a tag
- There should be a method that handles the start of every end tag

tagName

- The tag name must be a tag defined in the DTD of the `XMLParser` annotation

Table 4.2: Constraints for SaxSpoon Annotations

`XMLParser`

- `XMLParser` *target* `Class`
- `XMLParser` *type* `ContentHandler`

`HandlesStartTag`

- `HandlesStartTag` *target* `Method`
- `HandlesStartTag` *type* `void`
- `HandlesStartTag` *inside* `XMLParser`
- `HandlesStartTag`.tagName *inside unique* `XMLParser`
- `HandlesStartTag` *prohibits* `HandlesEndTag`
- `HandlesStartTag`.tagName *refersTo* `HandlesEndTag`.tagName

`HandlesEndTag`

- `HandlesEndTag` *target* `Method`
- `HandlesEndTag` *type* `void`
- `HandlesEndTag` *inside* `XMLParser`
- `HandlesEndTag`.tagName *inside unique* `XMLParser`
- `HandlesEndTag` *prohibits* `HandlesStartTag`
- `HandlesEndTag`.tagName *refersTo* `HandlesStartTag`.tagName

Figure 4.3: Generic constraints applied to the SaxSpoon Annotation Framework

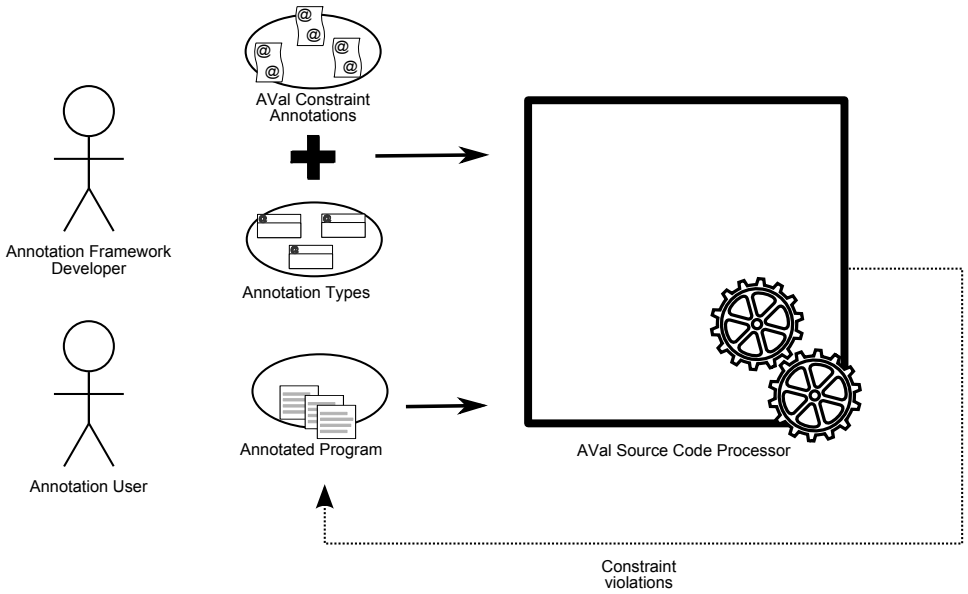


Figure 4.4: AVal Validation Process Flow

`Inside`	`UniqueInside`
`Prohibits`	`ProhibitsInside`
`Requires`	`RequiresInside`
`Unique`	`Modifier`
`RefersTo`	`AValTarget`
`URLValue`	`Type`
`RefersToAnnotatedElement`	

Table 4.3: AVal constraint annotation types

4.5 AVal: a (Meta) annotation framework to Specify Constraints

With the aim of providing an useful way to use generic constraints on annotation frameworks we have implemented a (meta) annotation framework that allows annotation framework developers to specify the constraints of their annotation types in a declarative way. Our tool, called AVal [NP07] – for Annotation Validation – consists of two parts: first, a set of annotations types that represents the generic constraints we have introduced in the past sections, and second, a source code processor that checks annotated programs in accordance to the semantics provided by the annotation framework developer.

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 [GJSB05] 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.

The process validating annotation frameworks using AVal is depicted in Figure 4.4. In it, the constraints of the annotation framework are codified as AVal annotations placed on the annotation framework itself. The AVal source code processor takes as input both the annotation framework and the annotated program, and reports constraint violations back to the annotated program in a format similar to that of the warnings and errors raised by a Java compiler. In the remainder of this section, we will introduce the annotation types provided by AVal, the architecture of the source code processor, and the way in which the AVal source code processor can be extended to deal with constraints other than those already provided.

4.5.1 AVal annotation constraints

As explained before, AVal provides a number of annotation types to specify constraints in annotation frameworks. This set of annotation types comes from the generic constraints defined before, having one annotation type per generic constraint. The annotation types offered by AVal is summarized in Table 4.3.

`Inside`, `Prohibits`, `Requires`, `Unique`, and `Type` are direct translations from the generic constraints introduced before. `UniqueInside`, `RequiresInside`, and `ProhibitsInside` represent the scope restriction/extension of *inside unique, inside requires* and *inside prohibits*.

In a similar way, `RefersTo` and `RefersToAnnotatedElement` are the annotation type representation of id references and type references. `AValTarget` represents the *target* constraint, renamed to avoid confusion with Java's own `Target` annotation. It takes as parameter AST nodes as represented by classes of the Spoon compile-time reflection API. Finally, `Modifier, Matches` and `URLValue` are additional constraints that serve as example of the extension points provided by AVal. `Modifier` takes as constraints the AST node on which the annotation is placed to have a Java modifier (`public, private, final`, etc.) `Matches` checks String typed elements against a regular expression given as parameter; and `URLValue` checks that a String element is a correctly formed URL.

4.5.2 AVal constraint composition

Generic constraints can be composed by a number of ways, as explained before. However, when representing these constraints as annotation types, we run into the limitations imposed by the programming language on the manner in which annotations are placed on code elements. The Java language specification does not allow more than one annotation of a given type to be placed on the same node. Because of this, it is impossible to, for example, express that a given annotation type requires several others by just annotating the type with several `Requires`. To allow the annotation framework developer to express these compositions of AVal annotations of the same type, *collection annotations* are needed. A collection annotation is an annotation type that serves as container to other annotations. If such a collection annotation exists, then the problem of stating that an annotation type *requires* several annotations is solved by creating a `RequiresAll` annotation type that has as sole element an array of `Requires` annotations and represents the conjunction of the *requires* constraints; a similar `RequiresAny` collection annotation could be constructed to represent the disjunction. AVal provides then a `*All` and `*Any` collection annotation for each of the basic constraint annotation types.

Having a different collection annotation for each constraint seems wasteful, and one is tempted to construct generic collection annotations `All` and `Any` that serve as containers to arbitrary annotations, this however is forbidden by the language as elements of annotation types cannot be just any annotation type. Because of this, the composition of several collection annotations into patterns of `*Any`s and `*All`s is impractical. This remains a limitation of the approach.

4.5.3 Example

To illustrate how AVal constraint annotations can be used to specify the constraints of an annotation framework, we go back to the example of SaxSpoon, and translate the generic constraints introduced in the example of the previous section to meta-annotated types. In the following Figure 4.5, the source code for each of the three annotations that compose SaxSpoon augmented with AVal annotations is shown.

Each of the annotation types defined in figure 4.5 uses AVal annotations to define the respective constraints. For the `XMLParser` annotation type, three constraints are defined: it must be placed on classes (line 1), the class on which it is placed must be a subtype of `ContentHandler` (line 2), and the *dtd* element must contain a string which is a valid URL

XMLParser.java

```
1 @AValTarget(CtClass.class)
2 @Type(ContentHandler.class)
3 public @interface XMLParser {
4   @URLValue
5   String dtd() default "";
6 }
```

HandlesStartTag.java

```
1 @Inside(XMLParser.class)
2 @Type(Void.class)
3 @Prohibits(HandlesEndTag.class)
4 public @interface HandlesStartTag {
5   @RefersTo(value=HandlesEndTag.class,
6             attribute = "tagName")
7   @UniqueInside
8   String tagName();
9 }
```

HandlesEndTag.java

```
1 @Inside(XMLParser.class)
2 @Type(Void.class)
3 @Prohibits(HandlesStartTag.class)
4 public @interface HandlesEndTag {
5   @RefersTo(value=HandlesStartTag.class,
6             attribute ="tagName")
7   @UniqueInside
8   String tagName();
9 }
```

Figure 4.5: SaxSpoon annotation types with AVal constraint annotations

(line 4). The `HandlesStartTag` and `HandlesEndTag` annotation types have similar constraints, and are therefore meta-annotated similarly: the `Inside` annotation in line 1 is used to state that the annotations can only be placed on sub-elements of that are annotated with `XMLParser`. They must be placed on methods with no return type (the `Type` annotation on line 2) and the *tagName* value must be unique on the subtree defined by the `Inside` annotation (line 7). Each of the `Handles` annotations prohibits the other one (line 3) and the `RefersTo` annotation checks that when a start tag event is handled, the corresponding end tag handler is defined (line 6).

As it is apparent, the use of constraint annotations to augment the definition of annotation types renders is a straightforward mapping from the generic constraints application of the previous section. The use of annotations to specify the constraints has the added value of making the annotation type's source code self documenting, in that their rules of use is clear from reading the source code alone.

Now that the annotation types provided by AVal have been explained, we can look into the way in which the actual constraint checking of an annotated program is carried out. This is explained in the next sections.

4.6 AVal Interpretation

Once the constraints of an annotation framework are defined using AVal annotations, the application developer can check whether these constraints are met in his application. The process of checking constraints on an annotated application consists on traversing the application's AST looking for annotations, once one is found, the definition of that annotation (i.e., its annotation type) is searched for AVal annotations. If any are found, the constraint that they represent is checked.

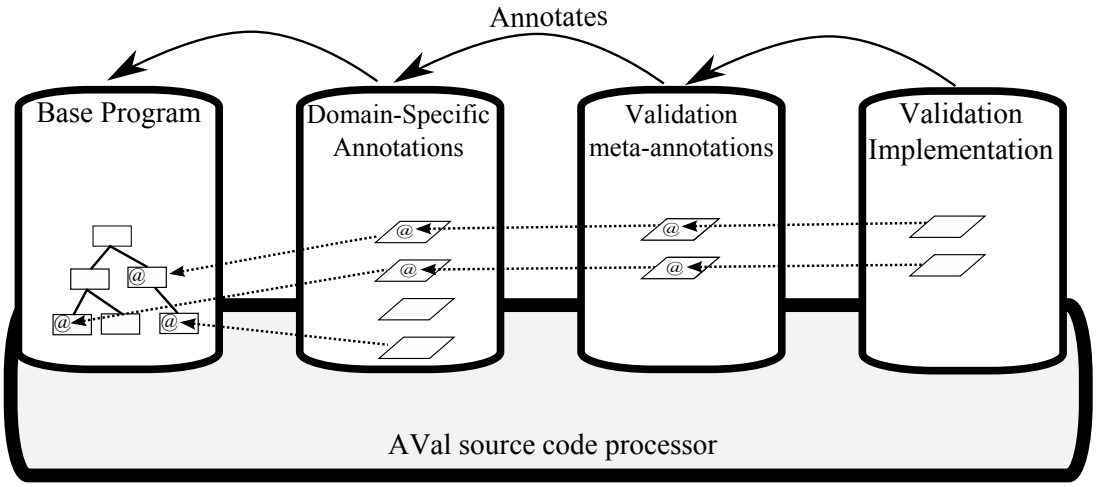


Figure 4.6: AVal Architecture

The process of constraint checking is depicted in figure 4.6. The base program's AST carries annotations defined in the domain-specific annotation framework (represented by dotted arrows). Some annotation types of the domain-specific annotation framework will carry AVal annotations, which are themselves tied to a corresponding implementation class. AVal, travels this *annotation chain* from an annotation in the base program, all the way to the implementation layer, recording each of the layers in a context object that will be used by the implementation to check the constraint. 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. Each of for layers is composed as follows:

Base program: The annotated program to be validated. Elements of the program are annotated by annotations defined on the Annotation Framework layer.

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

AVal Annotation Types: AVal annotations that encode the rules to validate domain specific annotations. Each annotation type represents a constraint, and is itself annotated with the class that is responsible for its implementation.

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

4.6.1 Extending validations

Even though the generic constraints defined so far cover many of the validation needs, there are cases in which domain specific constraints need to be defined. For these cases it is possible to extend the AVal's constraint annotations for a particular domain. AVal's architecture accommodates for these extensions, by adding annotation types and their corresponding implementations in the two upper layers; namely AVal annotation types and implementation. The implementation of a constraint annotation is a class that implements the `Validator` interface parametrized by the type of the annotation constraint. This interface defines a `check` method that is called whenever the validated annotation is found.

In order to carry out the constraint check, AVal provides the notion of a *Validation Point*. A validation point represents the context in which the constraint will be checked. It is composed of four parts: (1) the base program element in which the annotation to be checked is; (2) the annotation instance to be checked; (3) its annotation type, and finally (4) the AVal annotation constraint that is placed on the annotation type's definition. To illustrate this, imagine an annotation `@A("bar")` placed on a field (`int foo;`) within a class; further more, suppose that the annotation type of `public @interface A{}` carries an annotation constraint `@Constraint("baz")`. The corresponding validation point would be ⟨`int foo;`, `@A("bar")`, `public @interface A{}`, `@Constraint("baz")`⟩. In AVal, validation points are modeled as a class that contain the Spoon compile-time representations of each of its components; this gives the annotation framework developer access to the complete model of the base code, as well as to the model of his annotation framework. AVal uses then the full Java language to check constraints, which allows the annotation framework developer to delegate complex checks to specialized libraries and tools. 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.

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

public class NoUncheckedExceptionsValidator
              implements Validator<NoUncheckedExceptions>{
    public void check(ValidationPoint<NoUncheckedExceptions> 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.

4.6.2 Problem fixers and Error messages

Built-in generic constraints in AVal contain a number of special parameters, as defined in Section 7.1.2. 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 Annotation Framework. Each of these convenience elements are 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 annotation framework 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 SaxSpoon, 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 tag:

```
public @interface HandlesStartTag {
  @RefersTo(value=HandlesEndTag.class,
```

```
            attribute="tagName",
            message="No handler defined for the end of <?val> tag",
            severity=Severity.WARNING)
  String tagName;
}
```

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 SaxSpoon. 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 attached 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 *pre-defined* 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` without 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.

4.6.3 Library annotations

So far, in order to use AVal on a given Annotation Framework, the source code of the annotation types is necessary. Indeed, since the validation relies on meta-annotations, the AVal programmer must be able to add and remove annotations to the Annotation Framework. This, in principle, restricts the use of AVal to the annotation framework developers, since only they have access to the source code of the framework, and can modify it. Nevertheless, application programmers could also desire to enforce checks on the use of annotations in their programs; be it in response to internal coding guidelines, or because the annotation framework used does not provide an adequate constraint checker. To overcome this issue, in AVal it is possible to add validations to annotations for which the source code is not available 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. This kind of replacement is safe, since the modified version exists only during the validation phase. The replaced version is checked by the AVal source code processor so that it defines the same elements; doing this assures that the new meta-annotated version is syntactically equivalent to the old one.

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, like a method.

```
package dummy;

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

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

4.6.4 Eclipse Integration

AVal is integrated with the Eclipse IDE through the Spoon JDT plug-in[14]. This plug-in enables Spoon processors to be applied on a given Eclipse project each time it is compiled. By doing this, the relevant validations are applied seamlessly as dictated by the meta-annotations present on Annotation Frameworks 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 SaxSpoon program, is shown in Figure 4.7.

When an application developer wishes to use AVal to check the use of a given annotation framework whose annotation types carry AVal meta-annotation, he must load the spoonlet provided by AVal into the Spoon eclipse-plugin. The AVal plugin carries two processors: the dummy processor that takes care of the redefinition of annotations that do not carry AVal annotations (as defined in section 4.6.3), and the AVal processor that performs the validation of the application itself.

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

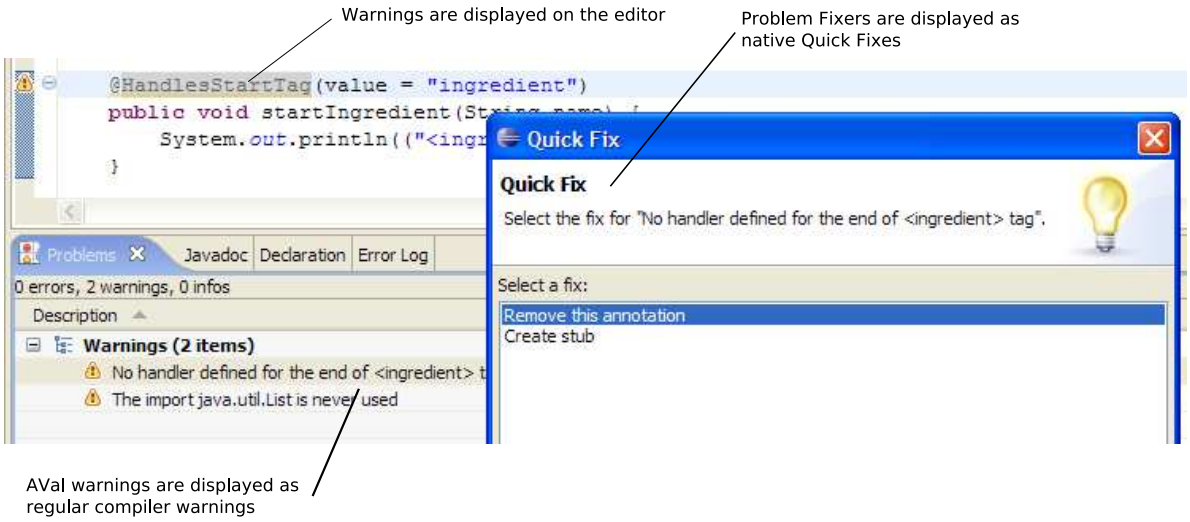


Figure 4.7: AVal integration with Eclipse IDE

4.7 Summary

In this chapter we have proposed a solution to the problem of defining and checking constraints for annotation frameworks (challenge III). We provide a classification of the kinds of constraints in function of whether they deal with annotation types or with the places in the code in which they lay. Based on this classification, we have proposed a number of reusable, declarative generic constraints which can be parametrized according to the needs of each specific annotation framework.

We have also presented a meta annotation framework that implements generic constraints called AVal. AVal provides two contributions: first a set of annotation types that represent the generic constraints we have identified. These annotation types are used by annotation framework developers to specify the constraints of their frameworks, which have the advantage of making explicit the semantics of the annotations. Second, AVal provides an extensible way to interpret these constraints, making it possible to add new, possibly domain-specific constraints that can leverage existing tools.

The use of meta-annotations, as described in this chapter, to specify constraints in annotation frameworks provides a twofold advantage. For the annotation framework developer, it is a declarative, reusable, extensible way to specify constraints, and for the annotation user, it provides timely constraint checking, and self documenting code.

Although ours is not the first approach that uses meta-annotations to check annotation type constraints [CM04]; our work goes further by defining constraints on both annotation's structure and the code on which they are placed. We have also based our implementation on the extensibility of the checking processor so that new AVal annotations can be defined (c.f. 4.6.1). Nevertheless, this extension mechanism remains tied to the Spoon API, which may be a high barrier to overcome for annotation framework developers not familiar with the tool. Generic constraints and AVal only address challenge

III, in the next chapter we present how to address the three remaining challenges by introducing the concept of *annotation models*.

5

Modeling Annotations

Contents

In chapter 3.4 we identified a number of challenges that arise when developing annotation frameworks. In the previous chapter we addressed challenge `III`; the definition and evaluation of constraints that make an annotated program valid. In this chapter we will address the remaining challenges, namely: `I` the representation of domain concepts as annotations, `II` the mapping of annotation types to code elements, and `IV` the reification of annotations present in a program.

***

The rest of the chapter is organized as follows: in the next sections we introduce the advantages of representing annotations with an object model (section 5.1) and introduce the concept of annotation models (section 7.1.3. In section 5.3 we present an annotation framework called ModelAn that allows the definition of annotation models by defining three annotation types: `Association`, `Default` and `Targets`; as well as a new way to define annotation framework constraints based on their annotation model, with an annotation type called `OCLConstraint`. In section 5.4 we present how annotation frameworks that use ModelAn are interpreted. Finally, a summary of this chapter is given in section 6.4.

5.1 Representing annotations as objects

As discussed previously, the definition and interpretation of annotations is complicated by the restrictions imposed by the Java language in their definition. Although annotations are not objects, several benefits can be obtained by representing them and manipulating them as if they were. Mapping concepts to objects is a much better understood process than to map them to annotation types (challenge `I`), and if annotations were objects, their reification (challenge `IV`) when processing annotated programs would be straightforward. It is for this reason, that we propose to represent annotations as objects.

A deeper look into the representation of annotations as objects brings up the following advantages:

- Seeing annotations as a separate object model would bring their definition closer to the domain model for the framework, since common modeling techniques rely mostly on OO concepts (UML, MOF for example),
- Such a representation would simplify the manipulation of annotations when processing them, since information such as the relationships between annotations would be preserved in the translation between the domain model and the annotation types,
- Finally, seeing annotations as an object model during an application's interpretation would allow a higher degree of abstraction over the manipulation of the AST of the interpreted program.

Having explored the advantages of representing annotations in a program as objects, the manner in which this will be achieved remains to be defined. This is the subject of the following section.

5.2 Annotation Models

Annotation types represent concepts from a given domain (see chapter 3.1). These concepts belong to a domain, which is the domain for which the annotation frameworks provides a solution. To faithfully represent annotations as objects, they must be consistent with the concepts of this domain, with their attributes, their relationships and their

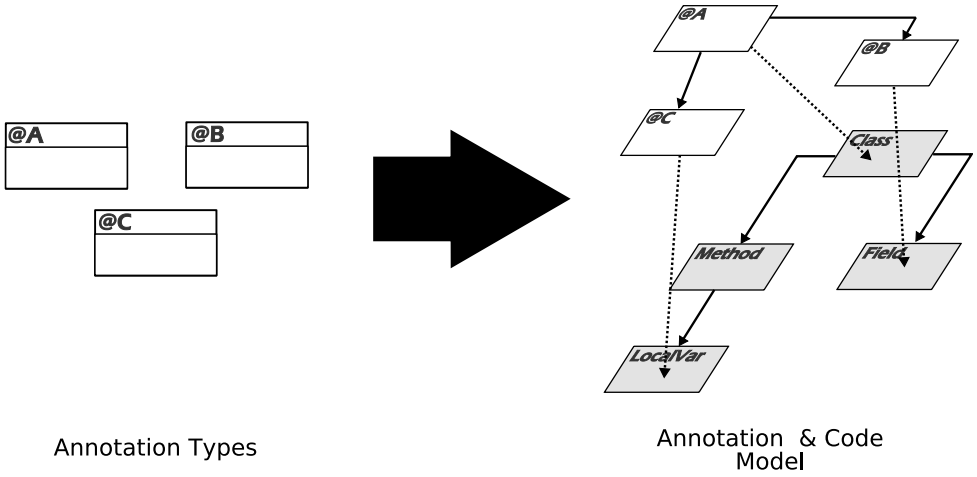


Figure 5.1: Annotation types and corresponding model

constraints. We will call the representation of the domain as defined by the annotation types an *annotation model*.

The goal of the annotation model is to reduce the gap between the annotation types and the domain model, by providing an intermediate representation which retains important information from the domain model, but remains close to the actual code that defines the annotation types.

An annotation model is then a representation of the domain of the annotation framework whose entities are those defined as annotation types. In addition to the concepts of the annotation types, relationships between them must be also represented. Finally, in order to have a complete representation, the annotation model must include a model of the code elements on which annotations are meant to be placed. This model of the code elements (*code model*) is the AST of the programming language, i.e., Java. On the left side of figure 5.1 an annotation framework composed of three annotation types: `A`, `B`, `C` that are meant to be placed on classes, fields and local variables respectably, is shown. In the right side of figure 5.1, the corresponding annotation model is shown. In it, the concepts from the annotation types are reproduced, and the relationships between them, which exist implicitly in the annotation types, are specified. Also, in gray, a part of the code model and the relations between it and the annotations are depicted by dashed lines.

Just as the annotation model represents the annotation types of a framework, it is possible to think that instances of the annotation model will represent an annotated program. The relationship between the set of annotation types, the annotation model, an annotated program, and instances of the annotation model are depicted in Figure 5.2. In it, the set of annotation types `A,B` and `C` is used in an application composed of two classes `Foo` and `Bar` (left side of the figure). The corresponding annotation and code model as well as the instance of it are depicted in the right side of the figure.

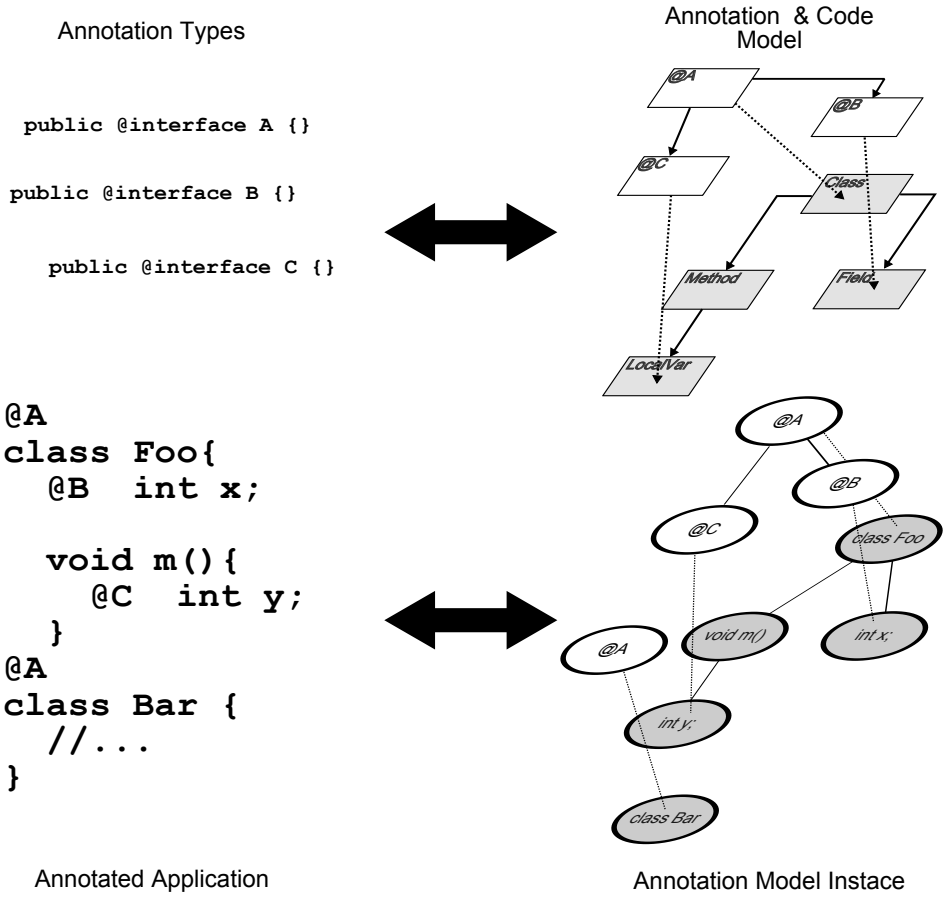


Figure 5.2: Annotation models are to annotation types like annotated programs are to annotation model instances.

5.2.1 Defining annotation models

In order to obtain an annotation model from a set of annotation types, and to instantiate that model from an annotated program, the annotation types must be augmented with the associations between them, default values and the constraints that will guarantee that an annotated program will be translated into a valid annotation model instance. Each of these additions will be discussed next.

5.2.2 Annotation associations

Associations between annotation entities in the annotation model describes semantic connections between the concepts that they represent. Since it is not possible to define such connections in the annotation types themselves due to Java's limitations, the associations must be externally defined. To model associations between annotation entities, we define an association as having a name and two ends: the owner of the association, and its target. This means that associations in the annotation model are binary and unidirectional. To keep the annotation model simple, we make no distinction for containment relations nor cardinality of the association's edges; both are handled via consistency constraints.

In order to be able to correctly create annotation model instances from annotated programs, the associations in the model must include information not only of the type of entities that are associated, but also of which particular instances of the entities participate. For example, consider our running example of SaxSpoon: the `XMLParser` annotation type is associated with the `HandlesStartTag` annotation. This alone is not enough to construct a model instance from an annotated program, since if several classes and methods carry `XMLParser` and `HandlesStartTag` annotations, it is impossible to know which `XMLParser` instances are associated with which `HandlesStartTag` instances. Because of this, in addition to the name and ends, an association in an annotation model must also define a *query*. The query, in the example, would specify that a `XMLParser` annotation placed on a given class is related only to `HandlesStartTag` annotations *placed on methods on that same class*[15] The inclusion of the query in the association is what allows the automatic instantiation of models from annotated programs.

5.2.3 Code associations

Annotations in applications are linked to the code elements on which they lay. Because of this, a relationship between annotation entities and code entities exists in the annotation model. Two kinds of annotation-code association are modeled: a concrete one between the annotation entity and the code element that represents the annotation, called `_annotation`; and an abstract one between the annotation entity and the code element on which the annotation is placed, called `target`.

The difference between these two relations is depicted in Figure 5.3. In it, the model instance for a class `Foo` annotated with `@A` is presented. In white, the annotation entity for the annotation type `A` has two relations, `target` going to code entities of type `class` and `_annotation` to the code entity, in gray, that represents the annotation in the AST.

[15] This query, formalized in OCL can be seen in section 5.2.6

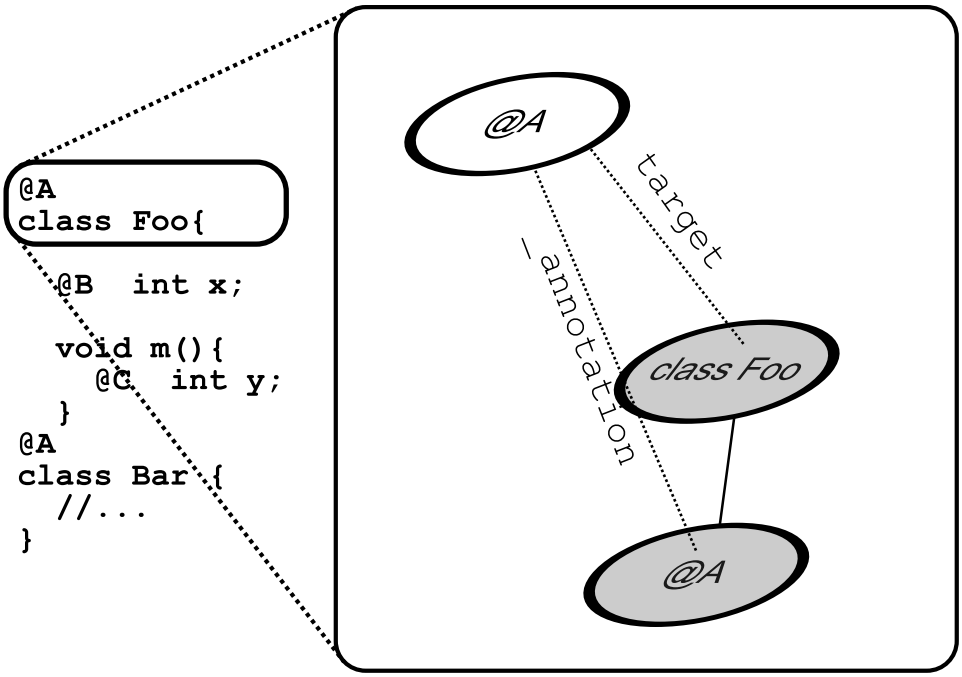


Figure 5.3: `_annotation` and `target` relations

5.2.4 Model consistency

As discussed in the previous chapter, annotation frameworks count with a number of constraints that govern the way in which it can be used. These constraints stem from the domain in which the annotation framework lies, and therefore, they must be reflected in the annotation model. With respect to the techniques exposed in the previous chapter, constraint checking at the annotation model level brings a number of advantages: constraint specification and checking are well research domains, and numerous tools and languages exist, allowing the annotation framework developer to leverage them. Also, the annotation model is closer to the domain model, and therefore, rule translation is simplified, by for example taking advantage of associations between annotation entities, which do not exist at the code level.

5.2.5 Default values

In annotation frameworks it is common to have the default value of an annotation type's element be derived from the code element on which it is placed. For example, in Fraclet [RPPM06] – an annotation framework for the Fractal [BCL+06] component model – an annotation type is used to mark a class as a component. By default, the name of the component is the simple name of the class on which the annotation lays. Although Java allows the definition of default values for annotation types, they must be compile-time constants, so expressing the default name for a Fraclet component annotation is impossible.

To remedy this, we propose to express the default value of annotation type's elements

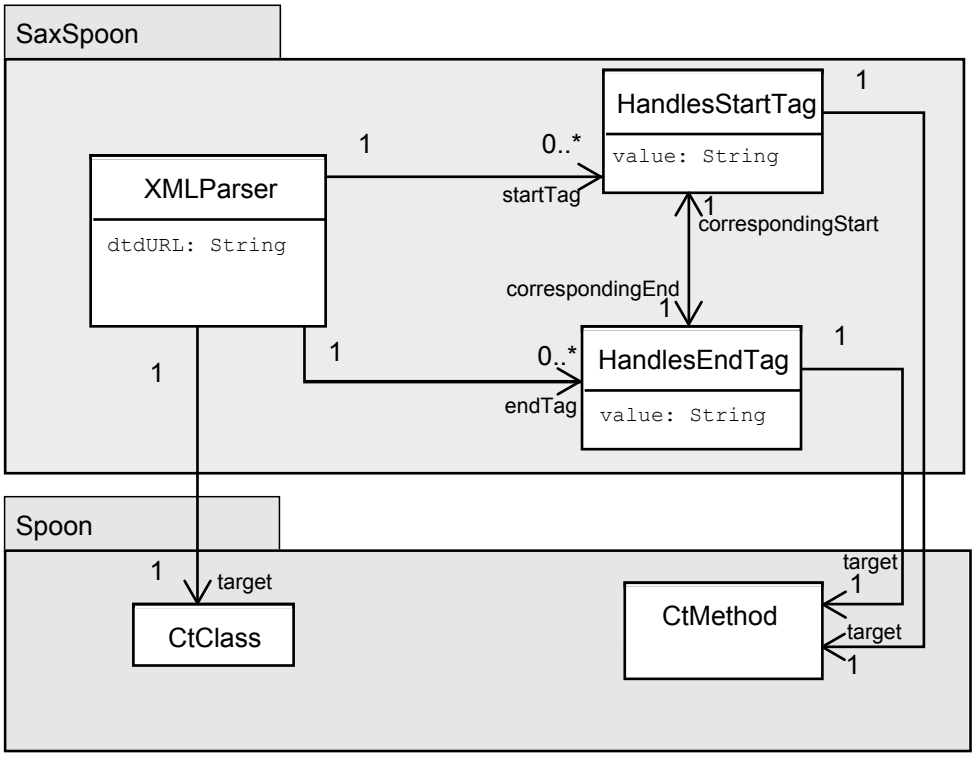


Figure 5.4: SaxSpoon annotation model

as a query on the annotation model for the framework. This provides an additional tool to the annotation framework developer, and further reduces the semantic gap between the domain of the framework and the annotation model.

5.2.6 Example

We will illustrate the annotation model definition process by defining one for the SaxSpoon annotation framework introduced in chapter 3.2. There are three annotation types in the framework: `XMLParser`, `HandlesStartTag` and `HandlesEndTag` that correspond to the concepts of handlers of the start and end tags of an XML document and their corresponding container.

SaxSpoon Annotation Model

The annotation model for SaxSpoon is depicted in figure 5.4. In the model, two packages are presented, one representing the annotation and the other, the code model. Each annotation type is translated into a classifier, and each annotation type element into an attribute.

Associations

The annotation entities in the model are linked with three associations: `start` and `end` denote the containment relation between a `XMLParser` annotation and its handlers; while

the `corresponding` association links handlers for a given tag (for example the start and end handler for a `recipe` tag). `Target` associations link each of the entities to their intended nodes in the AST of a program. In table 5.1, the queries expressed as Object Constraint Language (*OCL*) queries, that complete the definition of the associations are presented. These queries give the semantics of the associations by, for example, stating to which XMLParser is a HandlesStartTag annotation linked to.

```
startTag
  Context: XMLParser::startTag : Set(HandlesStartTag)
  HandlesStartTag.allInstances()->select(hst|self.target.Methods->includes(hst.target))
endTag
  Context: XMLParser::endTag : Set(HandlesEndTag)
  HandlesEndTag.allInstances()->select(het|self.target.Methods->includes(hst.target))
correspondingEnd
  Context: HandlesStartTag::correspondingEnd : Set(HandlesEndTag)
  HandlesEndTag.allInstances()->select(handler|handler.value = self.value)
correspondingStart
  Context: HandlesEndTag::correspondingStart : Set(HandlesStartTag)
  HandlesStartTag.allInstances()->select(handler|handler.value = self.value)
```

Table 5.1: SaxSpoon Association Queries

The queries are executed in the context of the owner of the association; in the case of `start` and `end` the owner is `XMLParser`, while for `correspondingEnd` the owner is `HandlesStartTag` and `correspondingStart` the owner is `HandlesEndTag`. The context of the query determines the type of the `self` pseudo-variable.

The `start` association query selects as participants of the association those instances of `HandlesStartTag` whose target method is one of the methods of the class which is targeted by the current `XMLParser` annotation (represented by `self`). For the `end` association, a similar query is used. For the `correspondingEnd`, the query selects the instance of `HandlesEndTag` that has the same value as the `HandlesStartTag` annotation represented by self; a similar query is used for the `correspondingStart` association.

Model Consistency

Having defined the annotation model for SaxSpoon, it is possible now to express the constraints that validate annotated programs as being correct SaxSpoon programs using the model. This is done via OCL invariants that reason on both the annotation and code model. Each of the constraints identified for the framework is specified in table 5.2.

Compared with the use of AVal for defining constraints, using OCL expressions over the annotation model has several advantages: it is a uniform specification, since all constraints are defined using the same language, as opposed to a mix of annotations and Java; the semantics of the constraints are well understood since they rely on OCL and the annotation and code model; and finally, they are composable in a simple way, since first order logic is provided by the OCL. Nevertheless, restricting the constraints to OCL brings up a number of problems, all of them rooted in the fact that OCL expressions can

`XMLParser`
▷Should be placed on classes that implement `ContentHandler`
`Context: XMLParser`
`self.target.SuperInterfaces->collect(sp|sp.SimpleName='ContentHandler').notEmpty()`[a]
▷There can only be one Start method handler per tag
`Context: XMLParser`
`self.start->isUnique(value)`
▷There can only be one End method handler per tag
`Context: XMLParser`
`self.end->isUnique(value)`
`HandlesStartTag`
▷The method should be of return type `void`
`Context: HandlesStartTag`
`self.target.Type.SimpleName = 'void'`
▷Methods parameters should all be of type String
`Context: HandlesStartTag`
`self.target.Method.Parameters->forAll(p|p.Type.SimpleName = 'String')`
▷A method cannot handle the start and end of a tag
`Context: HandlesStartTag`
`HandlesEndTag.allInstances()->select(het| het.target = self.target).isEmpty()`
`HandlesEndTag`
▷The method should be of return type `void`
`Context: HandlesEndTag`
`self.target.Type.SimpleName = 'void'`
▷The method should have a single parameter of type `String`
`Context: HandlesEndTag`
`self.target.Method.Parameters->forAll(p|p.Type.SimpleName = 'String') and`
`self.target.Method.Parameters->size() = 1`

[a] with the model alone it is not possible to check if the interface is implemented indirectly

Table 5.2: SaxSpoon Constraints

only reason on modeled entities. For example, the constraints dealing with Java types, like classes annotated with `XMLParser` implementing `ContentHandler`, must be done using the string representation of the class, since the type hierarchy of the annotated application is not modeled and cannot be modeled at the same time as the annotation framework since it would render the model specific to that application. Second, constraints dealing with constructs that are not part of the annotation framework nor of the AST of the program must be also included in the model. In the case of the constraints for the parameters of the methods marked `HandlesStart/EndTag`, a knowledge of the DTD of the XML documents on which the application will operate is needed. Without it, it is impossible, using OCL, to check the constraint that states that the parameters of the method that handles a tag must have the same names as the attributes defined for that tag. In AVal, both problems are avoided by implementing the constraint checking in Java. Since AVal constraints are implemented using Spoon they have access to both the AST and the types of the analyzed program, therefore performing checks that require knowledge of the types defined in the program is not an issue. Also, as AVal checkers have access to third-party libraries, they can delegate to them the analysis of non-Java artifacts, for example the validation of a DTD.

In the next section, we show a tool chain that allows the definition and instantiation of annotation models presented in this section using a dedicated annotation framework called ModelAn.

5.3 ModelAn: Annotation framework for Annotation Model Definition

In this section we present an annotation framework for the definition of annotation models from the source code of an annotation framework called ModelAn. The approach taken by ModelAn is similar to that of AVal (chapter 4.5): using annotations, the definition of the annotation types is augmented with the information necessary to construct an annotation model; namely the associations of which the annotation type participates, the default values for its elements, and the constraints to guaranty the consistency of instances of that model. Even though ModelAn takes a *code-directed* approach to defining annotation models, other approaches to the construction of annotation models are discussed at the end of this section. We start by introducing the annotation types that ModelAn offers in the next section.

5.3.1 Model definition

The annotation model is extracted from the 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. The model is then augmented by the annotation framework

developer using three annotations on the annotation types: `Association`, `DefaultValue` and `Targets`

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 defines the associations on the annotation model. For example, in the SaxSpoon annotation framework, there is a relation between a XMLParser and its start and end handlers. Therefore, the definition of the XMLParser annotation type would be as follows:

```
@Association(name = "Start",
             type = HandlesStart.class,
             query =
        "HandlesStart.allInstances()->
                    select(self.target.Methods->includes(target))")
public @interface XMLParser {
        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 `XMLParser`. Hence, this query constructively defines the relation *start*. A similar construction is used to define the relation between `XMLParser` and `HandlesEnd`

Default Value

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 value();

}
```

Targets

As discussed in section 5.2.3, just as relations between annotation entities are qualified by a query, relations between the annotation entities and code elements (i.e., the `target` relation) can also be qualified by queries. To this end, ModelAn offers a `Targets` annotations that allows the annotation framework developer to *auto-annotate* code entities

that correspond to a given OCL query. Suppose, for example, that in SaxSpoon, the developer would like to annotate all classes that directly implement `ContentHandler` with the `XMLParser` annotation. To do this, the definition of the `XMLParser` annotation would be meta-annotated as follows:

```
@Targets("CtClass.allInstances()->
    select(c |c.SuperInterface->
          any(si|si.SimpleName = 'ContentHandler') <> null)")
public @interface XMLParser {
        String dtdURL() default "";
}
```

5.3.2 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.

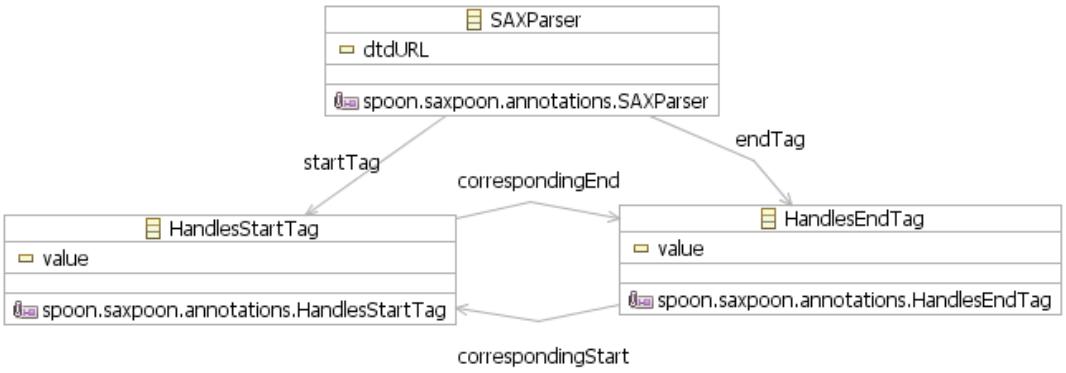
OCL expressions placed on annotation types can use the associations defined by the `Association` annotation to express the constraints of the annotation framework. The `OCLConstraint` annotation is an AVal annotation (see chapter 4.5) that defines a single property that contains the expression itself. 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 value();
}
```

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.

In figure 5.5, the annotation model of SaxSpoon and its corresponding annotation types marked with ModelAn annotations are shown.

The `Targets` annotation in conjunction with the `DefaultValue` annotation and the `OCLConstraint` annotation can provide interesting possibilities. Suppose that annotations are used to gather metrics of a system, one of them being the number of methods of classes in a given package `P`. Further more, because of design restrictions, classes with a large number of




```
@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 XMLParser {
        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();
}
```

Figure 5.5: SAXspoon Ecore Model and Annotated types

methods, five, are discouraged. By defining an annotation type `NOM`, and using the `Targets` and `DefaultValue` annotations, it is possible to encode the interpretation of the annotation in the annotation's definition itself:

```
1 @AValTarget(CtClass.class)
2 @Targets("CtClass.allInstances()->(c |c.Parent.SimpleName = 'P')
3 @OCLConstraint("self.value < 5")
4 public @interface NOM{
5   @DefaultValue("self.Methods->size()")
6   int value();
7 }
```

In line 1 the `Target` annotation states that this annotation is supposed to go on classes. In line 2, it is stated the kind of classes on which this annotation will go (that is classes on the package `P`). In line 5 the `DefaultValue` annotation states that the value of the element of the annotation is, precisely, the number of methods of the class on which this annotation lays. By processing this annotation definition with its interpretation engine, all of the classes belonging to the package `P` will carry the annotation `NOM` with the number of methods that they have. Finally, the `OCLConstraint` annotation in line 2, will instruct ModelAn's interpretation engine to raise an error on `NOM`s annotations that have a value greater than five: that is on those classes with more than five methods.

5.3.3 Other means of model definition

So far, `Association`, `DefaultValue` and `OCLConstraint` annotations have been used to define annotation models. There are however other means to specify the annotation models; we present two: using AVal constraints to implicitly define the annotation model, and explicitly defining the model using traditional modeling techniques.

Using AVal for model definition

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. Therefore, the `Inside` constraint actually represents both an association, and a constraint over it.

We have identified three AVal constraints that induce relationships in the annotation model: `Inside`, `Requires` and `RefersTo`:

Inside As already mentioned, stating that an annotation is constrained to be inside another one's scope implies a relationship. If the annotation developer states that `A inside B`, in the annotation model, an association going from `B` to `A` will be constructed with a name `N`. The query of the association will be

```
B.allInstances()->select(b | self.target.Methods->includes(b.target))
```

supposing that A's target is `class` and B's target is method. And, since `Inside` also implies a constraint the following expression

```
A.allInstances()->forAll(a|B.allInstances()->exist(b|a.N = b))
```

will be added to B's invariant, stating that no instance of A can exist if it is not related to an instance of A by the N association.

Requires When an annotation type A requires another one B, this means that the use of A on a given element makes no sense if it is not already annotated with B. Because of this, `requires` constraints imply an optional association. Supposing that the constraint `A requires B` exists on an annotation framework, then an association with name N going from B to A with the following query can be constructed.

```
A.allInstances()->select(a|a.target = self.target)
```

As the generic constraint must also be defined, the following expression is added to the invariant of B

```
B.allInstances()->forAll(b| A.allInstances()->exist(a| a.N = b)
```

As with the `Inside` constraint, this constraint states that all instances of B must participate in the association defined by the `Requires` constraint.

RefersTo These kinds of constraints are the ones that most strongly imply associations between annotations. In chapter 7.1.2, two kinds of `refersTo` constraints are introduced: id references and type references. In the first kind, an annotation A refers to an annotation B by the value of one of their elements v. In this case, an association going from A to B with the following query is constructed:

```
B.allInstances()->select(b| b.v = self.v)
```

In the case of type references, an annotation A has an element v that points to a Java type that carries an annotation B. In this case, the association going from A to B has the following query:

```
B.allInstances()->select(b| b.target.SimpleName = self.v.SimpleName)
```

In both cases, as with the previous AVal annotations, the constraint in the annotation model is that all instances of B must participate in an association:

```
B.allInstances()->forAll(b| A.allInstances()->exists(a| a.N = b)
```

Manual Model Construction

It is also possible to define the annotation model without resorting to annotating the annotation types of the framework, by creating the model *from scratch*. In this case, a modeling tool is used to construct the annotation model, creating by hand each of the entities that represent the annotation type, and using the tool's facilities to construct the

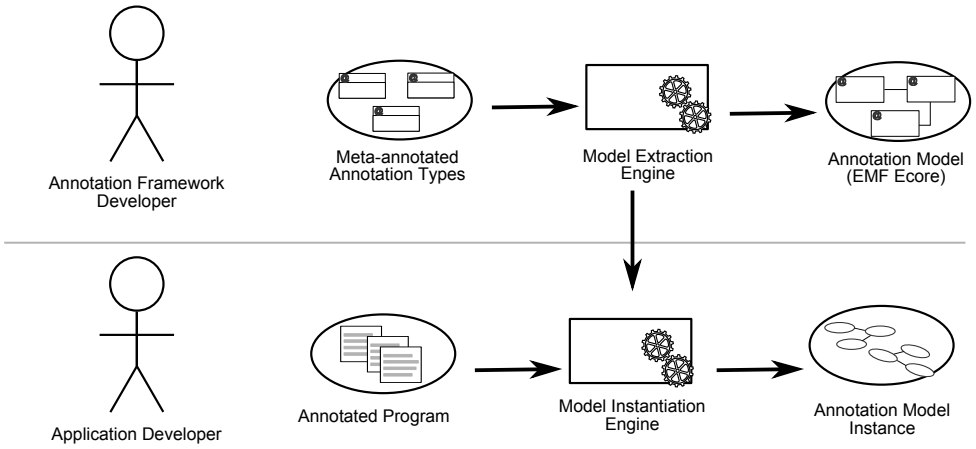


Figure 5.6: ModelAn interpretation flow

associations between them and between the annotation entities and the code model. This model, if it is to be integrated with the tool chains that extract annotation models from ModelAn or AVal annotations, must be consistent with the way in which the annotation entities are defined in this chapter; that is, the annotation entities must have the same name and attributes as the annotation types, the associations must carry queries, and links to the target and _annotation must exist for each annotation entity.

It is, of course, not a requirement that the annotation model be created by hand. It is possible to imagine that the annotation model would be the last step in a model transformation chain that start from a higher-level model. Such a transformation chain would, however, fall outside of the scope of this work.

5.4 ModelAn: Model Extraction

As presented in the previous section, ModelAn defines a number of annotation types to define the annotation model of an annotation framework. In this section we present how these annotations are interpreted in order to obtain an annotation model. The interpretation of ModelAn annotations is composed of three stages: the model construction from the annotation types, the generation of source code processors that will instantiate the annotation model, and the instantiation of the model itself. The interpretation of ModelAn annotations relies on several libraries. First of all, Spoon [PNP06] is used as an annotation processor and code generator. The Eclipse Modeling Framework is used to describe the annotation model, while SpoonEMF [Bar06] is used as a code model.

The general process for the interpretation of ModelAn–annotated frameworks is depicted in Figure 5.6. First, the annotation framework developer uses ModelAn annotations to meta-annotate its framework. The meta-annotated types are fed into the model extraction engine. From the model extraction engine, two artifacts are produced: an EMF Ecore model that represents the annotation model (section 5.4.1) and a model instantiation engine (section 5.4.2). The application developer then inputs the program, annotated with the annotation types of the framework, to the model instantiation engine produced in

the previous phase. By analyzing the annotated program, the model instantiation engine produces an instance of the annotation model (section 5.4.3).

Each of the phases of the interpretation of ModelAn is discussed in detail below.

5.4.1 Model construction

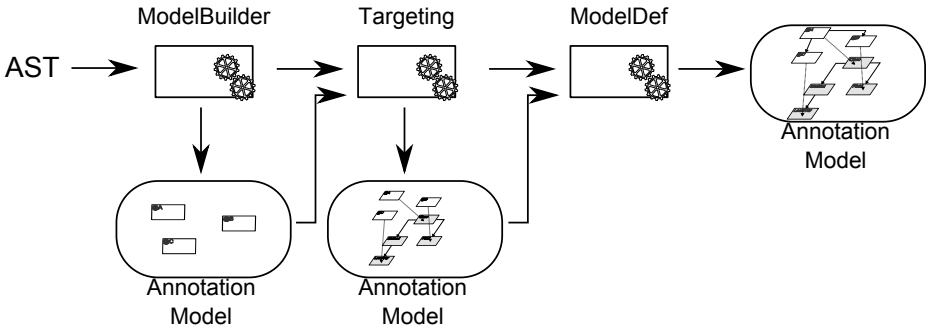


Figure 5.7: Model Construction

The construction of the annotation model is performed in several steps by Spoon source code processors. Each step contributes to the annotation model different features. figure

The first processor, called `ModelBuilder` is in charge of constructing entities of the annotation model. It starts by first creating the Ecore file that will contain the annotation model. Then, the processor visits each annotation type definition, creating an EClassifier (Ecore's classifiers) with the same name as the annotation type. Each of the elements of the annotation type is then added to the EClassifier, converting them to attributes for elements whose type are primitive types or enumerations, and as associations when the type of the element is an annotation type. The package structure that contains each annotation type is also reproduced in the annotation model. Finally, the `_annotation` association representing a reference to the AST representation of the annotation is added to the model. Since associations are constructed between annotation types and elements which are annotation types, a second pass over the EClassifiers, fixing dangling references is made. At the end of the processing round made by `ModelBuilder`, all of the annotation types are translated to EClassifiers. The annotations are then saved to an Ecore file, and kept in an in-memory dictionary for the subsequent processors.

The second processor, `TargetingProcessor` will create the `target` relations between the annotation entities in the model and their corresponding code elements. The target for an annotation type is taken from two sources: Java's `Target`, or AVal's `AValTarget` meta-annotations. If neither meta-annotations are present for a given annotation type, the `target` relation will point to the root type of the code model; in the case of SpoonEMF, this will be `CtElement`. If annotation type targets several code elements (for example, fields and local variables), several target relations will be created.

The third processor in the model extraction chain, `ModelDefProcessor` will be in charge of constructing associations between the annotation types as directed by the `Association`

annotations. The processor traverses the EClassifiers in the model, and for each one, it finds the corresponding annotation type (via the directory created by the `ModelBuilder`), and constructs EReferences for each `Association` annotation found in it. The query of the association is reproduced in the model as an EAnnotation on the reference. This processor also reproduces the default value information placed on the annotation types using the `DefaultValue` annotation on the model as EAnnotations on the attributes of the corresponding EClassifiers. At the end of the processing round, the annotation model will be complete. Note that since all the information present on the ModelAn annotation in the original annotation types is reproduced on the model, there is no further need for them. Because of this, it is possible to construct model instances from applications that use a version of the annotation framework that does not carry ModelAn annotations; this is important for the distribution of ModelAn-based processing tools.

Finally, a fourth optional processor, `AValModelBuilder`, can be applied to interpret AVal annotations to add associations to the annotation model as described in section 5.3.3.0. This processor acts in a similar way to the `ModelDefProcessor`.

In summary, in this phase, annotation types that carry ModelAn or AVal annotations is analyzed, and an Ecore file containing the annotation model is produced. The model extraction process is performed by four processors totaling close to 350 lines of code. The following phase will generate a source code processor that will take an annotated program and produce an instance of the annotation model.

5.4.2 Instantiator generation

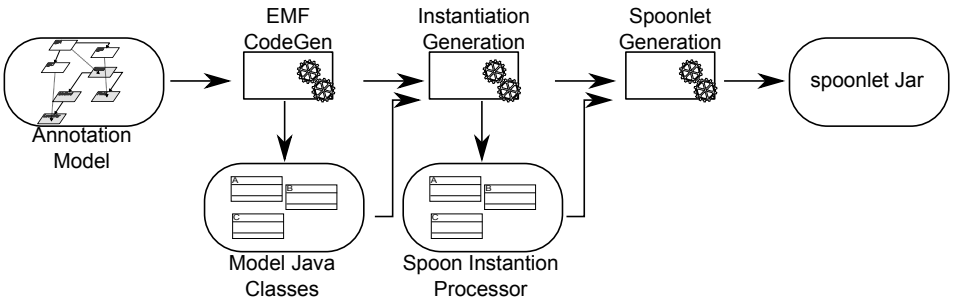


Figure 5.8: Instanciator Construction

In this phase, a Spoon source code processor to instantiate the annotation model produced in the previous phase is generated. The process is summarized in Figure 5.8. In addition to this, a set of Java classes that reify the annotation model are produced. This phase uses the EMF code generation facilities, as well as Spoon to analyze and generate the processor. This phase is comprised of three steps: the generation of the Java classes that will reify the model, the generation of the Spoon processor that will instantiate these classes (and through them, the annotation model), and finally, the generation of a Spoonlet plug-in.

As a first step, EMF is used to generate the Java classes that represent the annotation model. This process will translate EClassifiers to Classes, attributes to fields, and it will

generate getter and setter methods for the associations. The use of EMF to generate the classes brings as added benefit the set of tools that are a part of EMF, in particular, the OCL evaluation engine, which makes the implementation of the constraint checker straightforward.

Once the Java classes have been generated, the second step generates a source code processor that will instantiate them in accordance to an annotated program. The generation of the source code processor is carried out by the `InstantiationGeneration` Spoon processor. This processor starts by constructing the instantiation processor class. In essence, the instantiation processor will traverse annotations in a program, identifying the corresponding annotation entity in the model, and instantiating its Java class. The body of the instantiation processor is then a case statement to select the correct Java class for a given annotation. Once each case statement is added to the instantiation processor class, the `InstantiationGeneration` processor constructs a spoonlet descriptor.

Finally, in the last step, the EMF classes, the instantiation processor and the spoonlet descriptor are packaged into a Jar can then be used by Spoon to generate annotation model instances out of annotated programs.

By the end of the instantiator generation phase, the following artifacts are produced: a set of classes representing the reification of the annotations, a source code processor that given an annotated program generates an object graph that represents an instance of the annotation model, and a spoonlet jar that can be used by the Spoon plug-in to process programs. In total, the instantiator generation phase is implemented in 255 lines of code distributed in one processor and two source code templates.

5.4.3 Instance construction

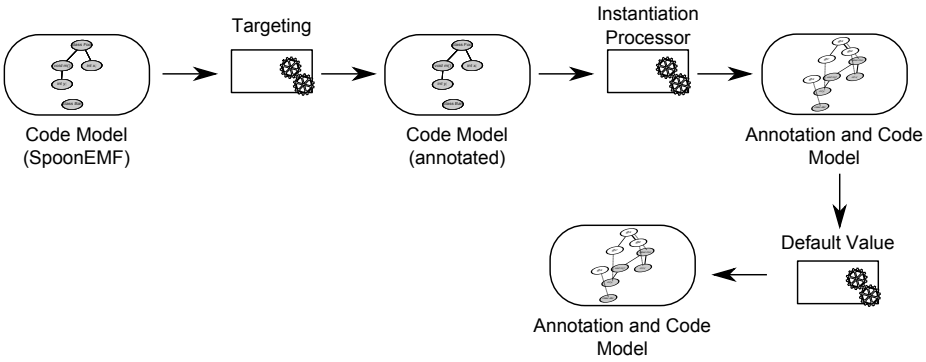


Figure 5.9: Instance Construction

The process of constructing an instance of a model from an input annotated program comprises several steps (Figure 5.9). First, a `Targeting` processor is in charge of placing annotations on the input program as directed by the `Targets` meta-annotations. This is done by evaluating the OCL queries of the `Targets` annotations and annotating the resulting code elements in the AST.

Following this, the instantiator processor generated in the previous phase is invoked. At the end of the processing round, the in-memory object graph of the annotation model's

```
1 public class OCLConstraintValidator
2      implements Validator<OCLConstraint> {

4 public void check(ValidationPoint<OCLConstraint> vp) {

6   //retrieve ocl expression to evaluate
7   String oclExpr = vp.getValAnnotation().value();

9   //Set up EMF OCL evaluation engine
10  //...
11  //get annotation reification from broker and put it in modObject
12  EClass c = modObject.eClass();

14  //Set context of OCL expression to current annotation entity
15  helper.setContext(c);

17  try {
18   //Evaluate OCL expression
19   OCLExpression exp = null;
20   exp = helper.createQuery(oclExpr);
21   boolean success = ocl.check(modObject,exp);
22   if(!success){
23      //Report error
24   }

26  } catch (ParserException e) {}
27 }

29 }
```

Figure 5.10: OCLConstraintValidator class responsible for annotation model consistency checking

instance is registered in a model broker that translates from annotations to their reified counterparts. In the last step, a `DefaultValueProcessor` will traverse the objects of the annotation model instance, evaluating the OCL queries that were defined in the `DefaultValue` annotations, and replacing the values of the attributes.

At the end of this phase, an object graph representing the annotation model instance that corresponds to the annotations present on the input program is obtained. This object graph is accessible through a `InstantiationBroker` that, given an annotation, produces the corresponding reification. The object graph can then be used to validate the constraints defined in the annotation model, to navigate the input program, or to aid in the interpretation of the annotation framework. In the following sections we detail the validation process and the visualization process.

5.4.4 Instance validation

The validation of the constraints of the annotation framework on the input program is performed using AVal (chapter 4.5). For this, the `OCLConstraint` annotation introduced in section 5.3.2 is annotated with `@Implementation(OCLConstraintValidator.class)`. The `OCLConstraintValidator` class will evaluate the OCL expression that represents the constraint, and report its possible violation.

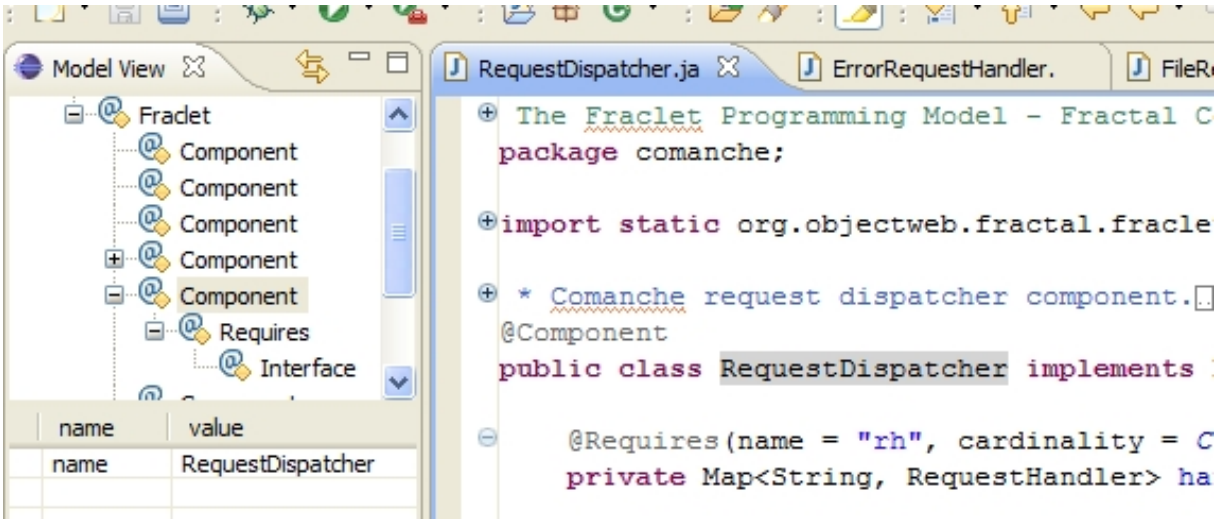


Figure 5.11: ModelAn Viewer Eclipse Plug-in for a Fraclet application

5.4.5 Instance visualization

In addition to constraint checking, the instance of the annotation model for a program can be used for program comprehension purposes. Given that the annotations used in a program represent a domain model (the annotation model) projected into the abstractions of the program (classes, methods, etc), the instance of the annotation model gives a domain-specific view on the application. For example, a complex application, using several annotation frameworks, would benefit from the direct visualization of each of the annotation model instances, since each one represents a facet of the application.

To aid the programmer, we have developed an Eclipse plug-in that provides a tree-based view of the annotation models in a given program. In figure 5.11, the ModelAn Viewer plug-in is used to visualize the use of the Fraclet annotation framework in a program. In a tree, all the annotation entities that represent components in the applications are displayed; drilling down on the tree, it is possible to see to which other annotation entities the current component is associated to. In the lower part of the view, there is a table with the attributes and values of the selected entity. When the programmer clicks on an annotation entity, the corresponding associated code entity is displayed in the IDE's editor, this is done via the `target` relations of annotation entities.

The viewer works by using the instantiation spoonlet discussed in section 5.4.2, and it formats the object graph that results in a tree structure. To do this, it is necessary to designate which elements will serve as roots to the tree, this is configured in the plug-in via an OCL query (figure 5.12).

5.5 Summary

In this chapter we have presented the notion of *annotation models*, and through it, we have proposed solutions to the three remaining challenges of annotation framework development. Annotation models provide a generic way of representing domain concepts as

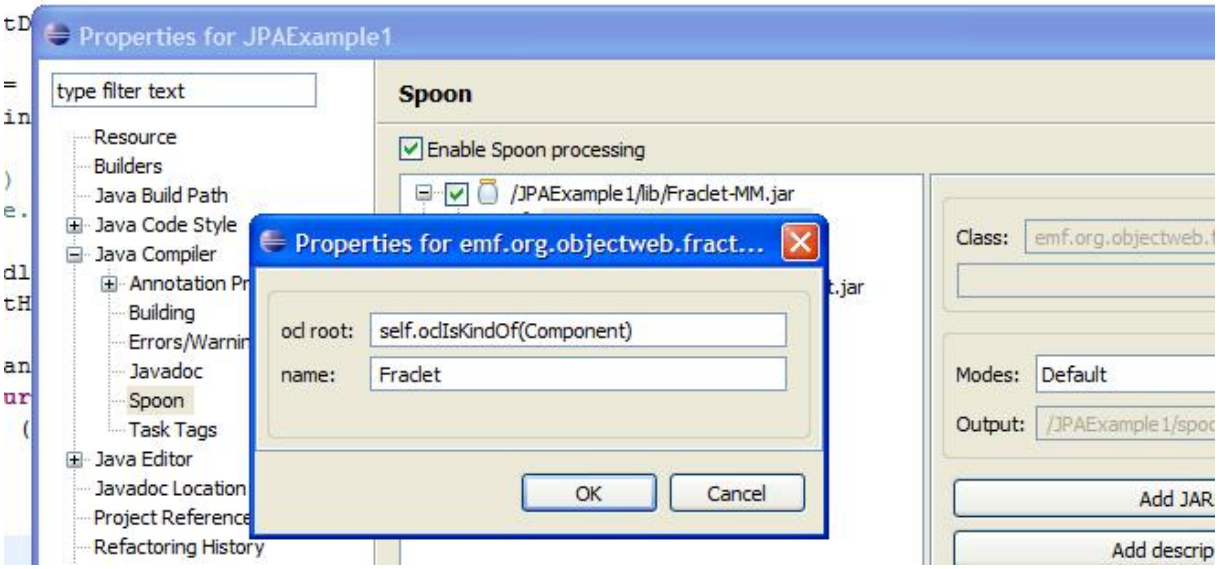


Figure 5.12: ModelAn Viewer Configuration

annotations (challenge I), the relationship between annotation types and code elements can be formally specified by the `target` relation (challenge II), and annotation models, when represented as Java objects, serve as a powerful reification of the annotations present in a program, since they provide direct access to associated annotations, and can be manipulated by EMF tools, such as OCL queries and visualization plug-ins.

In the following chapter, we show how the techniques proposed in this, and the previous chapters apply to three real-world annotation frameworks: the Fraclet annotation framework, the JSR 181 for web service development, and the Java Persistence API.

Part III

Validation

6

Case Studies

Contents

In this chapter we apply the techniques described in previous chapters 4.5 and 5.3 for defining, specifying and modeling three annotation frameworks: Fraclet, the JWS for specifying web services, and the Java Persistence API. The case studies will serve as both example, and validation of the usefulness of the proposed techniques.

The first case study, Fraclet, defines six annotations that represent basic notions in the Fractal component model. Fraclet-annotated classes contain the implementation of

primitive components by using a `Component` annotation to mark a class as a component, `Attribute` and `Requires` annotation to mark fields as containing attributes of the component in the former case, or instances of required interfaces in the latter. In addition to this, three annotations represent `Interfaces`, `Lifecycle` methods and `Controller` attributes.

The second case study, JWS, is a Java specification for the use of meta-data annotations to specify WebServices through seven annotations. Similar to Fraclet, classes annotated with `WebService` represent webService implementations. Services are provided through methods annotated with `WebMethod`, and configuration of the parameters and result of a web operation are given via the `WebParam` and `WebResult` annotations. Other annotations are used to specify the way the services are mapped to the underlying SOAP infrastructure.

The third and final case study, JPA, is the part of the EJB3 specification that defines the persistence of entity components. This is by far the largest of the case studies, totaling over sixty annotations. Annotations are used to map entities to tables, fields to columns, relationships between entities, queries, joint tables, etc. For brevity, in this document we discuss ten annotations of the sixty defined in the specification.

***

This chapter is divided into three parts, each dedicated to a case study: section 6.1 discusses Fraclet, an annotation framework for the Fractal component model; section 6.2 introduces the annotation framework for web service implementation JWS; and in section 6.3 we present the annotation framework for the Java Persistence API. Each of the case studies starts by giving an overview of the framework and an example of its use. This is followed by the constraints we identified, and the definition of the framework's annotation model. Finally the case study closes with an evaluation. After the case studies, the chapter is summarized in section 6.4.

6.1 Fraclet

Fraclet is an annotation framework for the Fractal component model [BCL+06]. In Fractal, components can be of two kinds: composite components, and primitive components, depending on whether they are composed of other components, or if they implement functionality themselves. Both types of components are defined by means of an architectural language called FractalADL[16]. In this language, the Fractal developer specifies the architecture of the application by defining components, composing primitive components into composite ones and defining the bindings between them and between the primitive components and their corresponding implementations in an underlying programming language (for example Java). The Fraclet annotation framework is used to ease the development of primitive components by embedding their definition into their Java implementation.

[16] http://fractal.objectweb.org/tutorials/adl/

6.1.1 Description

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 types defined by Fraclet. There are two implementations of Fraclet [RPPM06], one using XDoclet2 [WRO04], and the other one using Java5 annotations and Spoon annotation processor[17][PNP06]. We will study the latter in the remainder of this section.

Each of the annotation types defined in Fraclet are summarized in table 6.1. The `Component` annotation type marks a class as implementing a Fractal primitive component. `Component` defines two elements: the *name* of the component, and the set of `Interfaces` that the component offers. The required interfaces for the component are specified by tagging fields of the component class using the `Requires` annotation type. The `Requires` annotation type defines three elements: the *name* of the interface being required, the *cardinality* of the binding (i.e., whether the binding is a SINGLETON or a COLLECTION), and its *contingency* (i.e., whether the binding is MANDATORY or OPTIONAL). Attributes of the component are specified by means of the `Attribute` annotation type, that tags fields of component classes. The `Attribute` annotation type has a *name* for the attribute, as well as a default *value*, both specified as elements of the annotation type. Dependencies to the container (also known as controller in Fractal) of the component are specified by the `Controller` annotation, which has a single element that specifies the name of the service offered by the controller requested by the component. Finally, The `Lifecycle` annotation type is used to mark special methods of the component that are to be up-called when lifecycle-related events occur. The kind of event that each method handles is specified by the `Lifecycle`'s element *value*.

A Fraclet annotated application is not enough to completely describe a Fractal-based component architecture, since Fraclet only describes the implementation of individual components. Nevertheless, Fraclet provides a link between the application's architecture (described in the FractalADL language) and its implementation. Before the introduction of Fraclet, this link was implicit in the code of the primitive components.

6.1.2 Example application

In Figure 6.1, 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.

6.1.3 Constraints

The use of Fraclet on applications to define primitive components must adhere to a series of constraints for the interpretation engine to correctly generate the corresponding FractalADL and Java code. By discussing with the developers of Fraclet we have identified

[17] http://fractal.objectweb.org/tutorials/fraclet/

Annotation	**Location**	**Parameter**	**Description**
`Component`	Class	*name, provides*	Annotation to describe a Fractal component and its provided interfaces
`Interface`	Interface	*name, signature*	Annotation to describe a Fractal business interface.
`Attribute`	Field	*argument, value*	Annotation to describe an attribute of a Fractal component.
`Requires`	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 6.1: Overview of Fraclet annotations

the constraints presented in table 6.2. Based on the constraints, we will meta-annotate each of the annotation types defined in Fraclet with AVal generic constraints.

6.1.4 Annotation model

Having studied the Fraclet annotation framework, and having identified its constraints, it is now possible to define the Fraclet annotation model. For this, we use the AVal and ModelAn annotation types that embed in the source code of Fraclet's annotation types the associations and constraints of its annotation model. Each of the annotations is discussed below.

Component

```
1 @Retention(RUNTIME)
2 @AValTarget(CtClass.class)
3
4 @OCLConstraint(self.contains_Attribute->collect(name)"
5               +"union self.contains_Controller->collect(value)"
6               +"union self.contains_requires->collect(name)
7               +"isUnique(a |a)")

9 public @interface Component {
10   @Unique
11   @Default("self.target_CLASS.SimpleName")
12   String name() default EMPTY;

14   Interface[] provides() default {};
15 }
```

The target of the `Component` annotation type is specified using the `AValTarget` annotation in line 2, the AVal annotation is used because Java's `Target` annotation does not allow to

`Component`

- 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` annotation is placed.
- The names of the attributes, controllers and required interfaces must be unique to the component.

`Interface`

- An interface's name must be unique in the application. Since bindings between components and interfaces is made using the name of the interface, the name must be unique.
- When used inside a `Component` , Interface.signature must be implemented by the class
- Cannot be put on a class which is already annotated `Component`

`Attribute`

- Attribute fields are only allowed on classes marked as component
- A Field on a Component cannot be at the same time Attribute and Required
- Attributes can only be placed on fields that have a primitive type, because of the default value element.

`Requires`

- Requires fields are only allowed on classes marked as component
- A Field on a Component cannot be at the same time Attribute and Required
- 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.

`Lifecycle`

- Lifecycle methods are only allowed on classes marked as component
- Lifecycle methods cannot have parameters

`Controller`

- Controller fields are only allowed on classes marked as component

Table 6.2: Fraclet annotation's constraints

```
@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);
 }
}
```

Figure 6.1: Client Component Fraclet Implementation

make the distinction between classes and interfaces as targets, grouping them into a single TYPE target. In line 11, the default value for the name of the components is made to be the simple name of the class on which the annotations are to be placed. Finally, an `Unique` AVal annotation is used to state that the name of the components must be unique throughout the application (line 10).

In line 4, we constraint attributes, controllers and required ports of this component to have unique names.

Interface

```
1 @Retention(RUNTIME)
2 @AValTarget(CtInterface.class)
3
4 @OCLConstraint("self.target_INTERFACE.Superinterfaces->"
5     +"exists(i|i.name= self.signature.SimpleName)")
6 public @interface Interface {
7     @Unique
8     String name() default EMPTY;

10    Class<?> signature() default Constants.class;
11 }
```

The `Interface` annotation type is meant to be placed on Java interfaces only, as stated by the `AValTarget` annotation in line 2. The name of an `Interface` must be unique (line 7), and, when used as an element of a `Component` annotation, the type declared in the `signature` element must be implemented by the class. This is expressed in the `OCLConstraint` annotation in line 4.

Attribute

```
1 @Retention(RUNTIME)

3 @Prohibits(Requires.class)
4 @Target(FIELD)

6 @Inside(Component.class)
7
8 @OCLConstraint("self.target_FIELD.Type.isPrimitive()")
9 public @interface Attribute {

11   @Default("self.target_FIELD.SimpleName")
12   String name() default EMPTY;

14   String value() default EMPTY;
15 }
```

As fields cannot carry at the same time `Attribute` and `Requires` annotations, the `Attribute` annotation type prohibits the `Requires` annotation (line 3). The `Inside` annotation in line 6 is used to constraint `Attribute`s to classes annotated with `Component`, and to create the `contains_Attribute` association used in the definition of the `Component`'s constraints. Finally, the OCL expression in line 8 constraints the type of the fields on which an `Attribute` can be places to primitive types.

The name of the `Attribute` is made to be the simple name of the field on which it is placed (line 11).

Requires

```
1 @Retention(RUNTIME)
2 @Target(FIELD)
3
4 @Prohibits(Attribute.class)
5
6 @Inside(Component.class)
7
8 public @interface Requires {

10     @Default("self.target_FIELD.SimpleName")
11     String name() default EMPTY;

13     Cardinality cardinality() default SINGLETON;

15     Contingency contingency() default MANDATORY;

17 }
```

As with `Attribute`, `Requires` annotations must be placed with in fields of `Component` classes, and they prohibit the use of `Attribute` of the same target (lines 6, 2 and 4).

Controller

```
1 @Retention(RUNTIME)
2 @Target(FIELD)
3 @Inside(Component.class)
4 public @interface Controller {
5     String value() default "component";
6 }
```

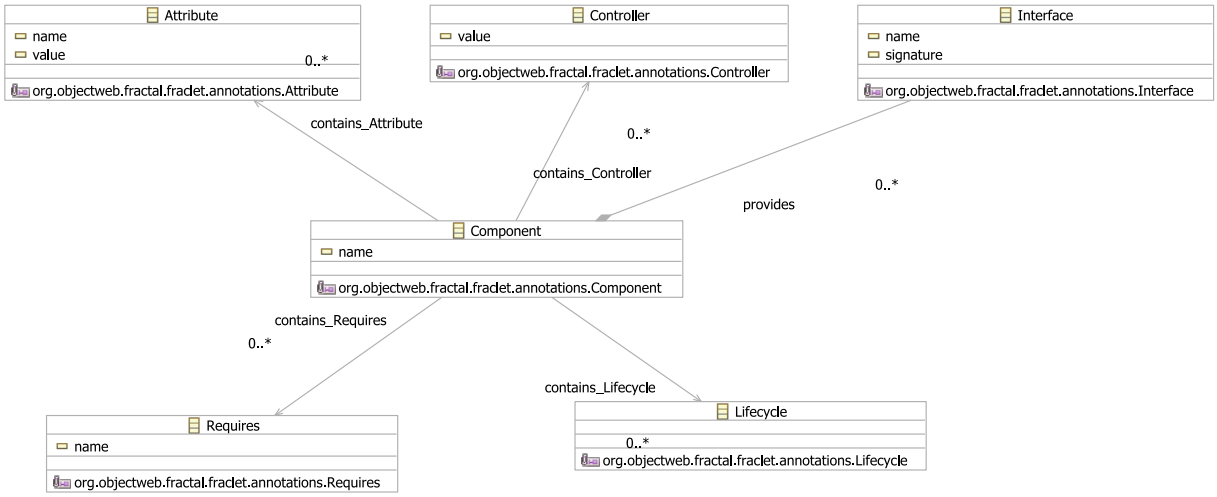


Figure 6.2: Fraclet Annotation Model

Controllers are meant to be placed on fields belonging to `Component` classes (lines 2 and 3).

Lifecycle

```
1 @Retention(RUNTIME)
2 @AValTarget(CtMethod.class)
3 @Inside(Component.class)
4
5 @OCLConstraint("self.target_METHOD.Parameters.isEmpty()")
6 public @interface Lifecycle {
7     Step value();
8 }
```

Life cycle methods must have no parameters (line 5) and must be placed inside of `Component` annotated classes (line 3).

Using the ModelAn tool-chain on the annotation types described above, the annotation model shown in figure 6.2 was produced.

6.1.5 Evaluation

There are several advantages of the use of AVal and ModelAn annotations on the Fraclet annotation framework. For the application developer, the specification of the framework is now clearer. The documentation for the annotation framework is incomplete or ambiguous in places (for example on the requirements of the `Lifecycle` annotation it is not clear if the method on which the annotation is placed can have parameters). We remedy this by embedding the constraints in the source code of the framework's interface; making in self-documented. In addition to this, the extraction of the annotation model (figure 6.2), aids in the understanding of the framework for users, since the way in which annotations relate to each other is now evident. For the annotation framework developer, in addition

to clarifying the semantics of each of the annotation types, ModelAn annotations reduce the burden of writing code to check the constraints, and, through the `Default` annotation, simplifies its interpretation.

In all of Fraclet's annotations, AVal annotations are used both to define constraints and to define associations. The `Inside(Component.class)` annotation in Fraclet's `Attribute`, `Controller`, `Interface`, `Requires` and `Lifecycle` annotations induces the `contains_` relations that are shown in figure 6.2. This rids the Fraclet developer of the definition of these associations. However, the developer must be familiar with the way in which ModelAn interprets the `Inside` annotation when using the associations defined by it, in particular, the sense of the association (which is inverse of the one defined by the `Inside` annotation) and the name of the association (`contains_` plus the name of the annotation).

The ModelAn tool-chain generated 1717 lines of code for the Fraclet annotation model. Of these, 219 lines correspond to the instantiation processor, while the rest (1498) correspond to the Java classes that represent the reification of the annotation types.

6.2 Java Web Services

The Java Web Service (*JWS*) [Zot05] is a specification for the description of web services using pure Java objects. The JWS defines a set of annotations and their mapping to the XML-Based *Web Service Description Language*. This specification is made to ease the development of web services in Java by merging the service's implementation with its definition. Web services are composed of a number of web methods that actually implement the services. The way in which the methods are invoked, their parameters and return values managed, are all defined using annotations in JWS applications.

6.2.1 Description

The JWS specification defines eight annotations for the implementation of web services. These annotations are summarized in table 6.3.

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 a *one-way* operation must have no return value.

6.2.2 Example application

To give a better idea of how the JWS annotations are used to define a web service, consider the following source code listing:

```
1 @WebService(targetNamespace="http://www.openuri.org/jsr181/WebParamExample")
2 @SOAPBinding(style=SOAPBinding.Style.RPC)
3 public class PingService {
4   @WebMethod(operationName = "PingOneWay")
5   @Oneway
6   public void ping(PingDocument ping) {
7   //...
8   }

10  @WebMethod(operationName = "PingTwoWay")
11  public void ping(
```

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
`SOAPBinding`	Class, Method	*style, use, parameter*	Specifies the mapping of the WebService onto the SOAP message protocol

Table 6.3: Overview of JWS annotations

```
12          @WebParam(mode=WebParam.Mode.INOUT)
13          PingDocumentHolder ping) {
14    //...
15    }

17    @WebMethod(operationName = "SecurePing")
18    @Oneway
19    public void ping(
20          PingDocument ping,
21          @WebParam(header=true)
22          String secHeader) {
23    //...
24    }
25  }
```

In it, the `PingService` class implements a web service of the same name which uses the `targetNamespace` defined by the URL in the `WebService` annotation in line 1. The web service provides three web methods (defined with `WebMethod` annotations in lines 4, 10 and 17); of these, the `PingOneway` and `SecurePing` web methods are oneway methods. All methods use RPC as invocation style, as specified by the `SOAPBinding` annotation in line 2.

6.2.3 Constraints

Constraints defined in the JWS specification are summarized in table 6.4. Using these constraints, and the description of each annotation type in the JWS specification we will define the framework's annotation model and based on it, use AVal and ModelAn annotations to specify the framework.

WebService

- The `wsdlLocation` element must be a URL.

WebMethod

- All methods of `WebService` classes are web methods
- Only methods belonging to `WebService` classes can be `WebMethods`
- If the `exclude` element is `true`, all other elements must remain empty.

Oneway

- Can only be placed on methods that carry the `WebMethod` annotation.
- Methods annotated with `Oneway` must be of type void, not throw checked exceptions and have no parameters whose mode are INOUT or OUT.
- Requires WebMethod

WebParam

- This annotation can only be placed on parameters of a method with the `WebMethod` annotation.
- The `name` element must be specified if the `WebMethod` operation is DOCUMENT style, the parameter style is BARE and the mode is OUT or INOUT
- If the `header` element is true, the type of the parameter must be a primitive type.

WebResult

- This annotation can only be placed on methods with the `WebMethod` annotation.
- If the `handler` element is true, then the type of the method must be primitive.

HandlerChain

- This annotation cannot be placed on methods nor fields.
- The `file` element can only contain strings which are valid URLs.

SOAPBinding

- Can be in methods only if the `style` element is DOCUMENT

Table 6.4: JWS annotation's constraints

6.2.4 Annotation model

We, again use ModelAn and AVal annotations to define the annotation model and its corresponding constraints for the JWS framework. The source code for each of the annotation types and their corresponding annotations are explained below.

WebService

```
1 @Target(ElementType.TYPE)
2 @Retention(RetentionPolicy.RUNTIME)

4 public @interface WebService {
5  @Default("self.target_TYPE.SimpleName")
6  String name() default "";

8  String targetNamespace() default "";

10  @Default("self.target_Type.SimpleName.concat('Service')")
11  String serviceName() default "";

13  @URLValue()
14  String wsdlLocation() default "";

16  String endpointInterface() default "";

18  @Default("self.name.concat('Port')")
19  String portName() default "";
20 }
```

The `WebService` annotation has only one constraint, that the `wsdlLocation` string element be a valid URL. To check this we use the `URLValue` (line 13) AVal annotation introduced in chapter 4.6.1. Three default values are used: the name of the webService is the simple name of the type on which it is placed, the name of the service is the concatenation of the name of the webService and "Service", and the name of the port is the concatenation of the name of the webService and "Port". Each of these default values is implemented by the `Default` annotations in lines 5, 10 and 18.

WebMethod

```
1 @Target (ElementType.METHOD)
2 @Retention(RetentionPolicy.RUNTIME)

4 @Inside(WebService.class)
5 @Associations({
6   @Association(name="service",type=WebService.class,
7  query="WebService.allInstances()->select(ws|ws.contains_WebMethod->contains(self))")

9   @Association(name="SOAPBindings",type=SOAPBinding.class,
10  query="SOAPBinding.allInstances()->select(s|s.target_METHOD = self.target_METHOD)"),
11 })

13 @OCLConstraint("self.exclude implies "+
14      "(self.action = '' and self.operationName = self.target_METHOD.SimpleName)")

16 @Targets("WebService.allInstances().target_Class.Methods")
17
```

```
18 public @interface WebMethod {

20   @Default("self.target_METHOD.SimpleName")
21   String operationName() default "";

23   String action() default "";

25   boolean exclude() default false;
26 }
```

`WebMethod`s are only valid on methods of `WebService` classes. Furthermore, all methods belonging to `WebService` classes are `WebMethod`s. The first constraint is realized by the `Inside` annotation in line 4. The second one is expressed by the `Targets` annotation in line 16. The value of the `Targets` annotation is an OCL expression that evaluates to all the methods belonging to a class annotated `WebService`. ModelAn annotation model instantiator will then annotate each of these methods with a `WebMethod` annotation using its default values. Another constraint states that the `exclude` element, when set to `true`, implies that no other element in the annotation can be set. This is expressed by the `OCLConstraint` annotation in line 13.

In addition to the constraints, we define two associations. The `service` association (line 6) serves as an inverse relation to `WebService` for the `contains_WebMethod` induced by the `Inside` in line4. The second association is the `SOAPBindings` that relates the WebMethod with its corresponding SOAP binding mapping (line 9).

Oneway

```
1 @Retention(RetentionPolicy.RUNTIME)
2 @Target({ElementType.METHOD})

4 @Requires(WebMethod.class)
5
6 @Association(name="WMethod",type=WebMethod.class,
7  query="WebMethod.allInstances()->select(wm|wm.optional_Oneway = self)")

9 @OCLConstraints({
10  @OCLConstraint("self.WMethod.contains_WebParam->"+
11     "forall(wp | wp.mode <> 'INOUT' and wp.mode <> 'OUT'"),
12  @OCLConstraint("self.target_METHOD.ThrownTypes->isEmpty()")
13 })
14 @Type(Void.class)
15 public @interface Oneway {

17 }
```

Methods annotated `Oneway` must also carry the `WebMethod` annotation, which is stated with the `Requires` annotation in line 4. In addition to this, `Oneway` method's parameters cannot be marked INOUT nor OUT, and they must not throw checked exceptions. To specify this, an association (inverse to that specified with the `Requires`) must be included (line 6). Using this `WMethod` association, OCL expressions to specify the constraints on the web method's parameters are defined in lines 11 and 12. Finally, `Oneway` web methods must have no return type (`void`), which is specified by AVal's `Type` annotation in line 14.

WebParam

```
1 @Target (ElementType.PARAMETER)
2 @Retention(RetentionPolicy.RUNTIME)

4 @Inside(WebMethod.class)
5 @Association(name="WMethod",type=WebMethod.class,
6  query="WebMethod.allInstances()->select(wm|wm.contains_WebParam.contains(self))")

8 @OCLConstraints({
9  @OCLConstraint("(self.WMethod.SOAPBindings.style = 'DOCUMENT'"
10      +" and self.WMethod.SOAPBindings.parameterStyle = 'BARE'"
11      + " and (self.mode = 'INOUT' or self.mode = 'OUT'))" +
12         " implies self.name <> ''"),

14  @OCLConstraint("self.header implies self.Parent.Type.isPrimitive()")
15 })

17 public @interface WebParam {
18         public enum Mode{IN, OUT, INOUT};
19         String name() default "";
20         String targetNamespace() default "";
21         Mode mode() default Mode.IN;
22         boolean header() default false;
23         String partName() default "";
24 }
```

The `WebParam` annotation is only valid in parameters for web methods (line 4). If the SOAPBinding style of the web method is DOCUMENT.BARE, and the parameter is either INOUT or OUT, the web parameter must be named, as stated in the `OCLConstraint` annotation in line 9. Also, if the web parameter is to be passed in the header of the message (`header` element is `true`), then the type of the method parameter must be a primitive type (as per constraint in line 14).

As with other annotation types, we define an association back to the containing element, in this case `WebMethod`, to simplify the OCL constraints. This association is defined in line 5.

WebResult

```
1 @Target (ElementType.METHOD)
2 @Retention(RetentionPolicy.RUNTIME)

4 @Requires(WebMethod.class)
5
6 @OCLConstraints({
7  @OCLConstraint("self.header implies self.target_METHOD.Type.isPrimitive()"),
8  @OCLConstraint("self.target_METHOD.Type.SimpleName <> 'void'")
9 })
10 public @interface WebResult {
11         String name() default "";
12         String targetNamespace() default "";
13         boolean header() default false;
14         String partName() default "";
15 }
```

We have three constraints for `WebResult`: First, it requires that the method carries the `WebMethod` annotation (line 4), as with the `WebParam` annotation. Second, if the result is mapped to the header of the message, the return type must be a primitive type (line 7).Third, the method must have a return type (line 8).

HandlerChain

```
1 @Target ({ElementType.TYPE, ElementType.METHOD, ElementType.FIELD})
2 @Retention(RetentionPolicy.RUNTIME)

4 @Requires(WebService.class)
5
6 @OCLConstraints({
7         @OCLConstraint("self.target_METHOD = null"),
8         @OCLConstraint("self.target_FIELD = null")
9 })
10 public @interface HandlerChain {

12  @URLValue
13  String file();

15  @Deprecated
16  String name() default "";
17 }
```

Although the `HandlerChain` annotation can target types, methods and fields (line 1), the specification states that it is only allowed to place it in types. The other two targets are defined so that the annotation type is compatible with other frameworks. To check this, two OCL constraints in lines 7 and 8 check the target of the annotation. In addition to this, `HandlerChain` annotations can only be placed on types that are already annotated with `WebService` (line 4).

The specification states also that the `name` element should not be used (therefore marking it `Deprecated`), and that the `file` element must be a valid URL (line 12).

SOAPBinding

```
1 @Target ({ElementType.TYPE, ElementType.METHOD})
2 @Retention(RetentionPolicy.RUNTIME)


5 @OCLConstraint("self.target_METHOD <> null implies self.style = 'DOCUMENT'"),
6 public @interface SOAPBinding {
7     public enum Style{DOCUMENT, RPC};
8     public enum Use {LITERAL, ENCODED};
9     public enum ParameterStyle{BARE, WRAPPED};
10    Style style() default Style.DOCUMENT;
11    Use use() default Use.LITERAL;
12    ParameterStyle parameterStyle() default ParameterStyle.WRAPPED;
13 }
```

Constraints dealing with the `SOAPBinding` annotation have already been defined in the `WebParam` and `WebMethod` annotation types. The last constraint states that the style DOCUMENT is required for annotations placed on methods.

Using the annotated types defined above, the ModelAn model extraction engine generates the model depicted in figure 6.3. In it, both explicit associations expressed via the `Association` annotation and implicit associations using the AVal annotations are shown. For example, `Oneway` is marked with `Requires(WebMethod.class)`, ModelAn interprets this as an implicit relation between `WebMethod` and `Oneway` called `optional_Oneway`. However, in order to

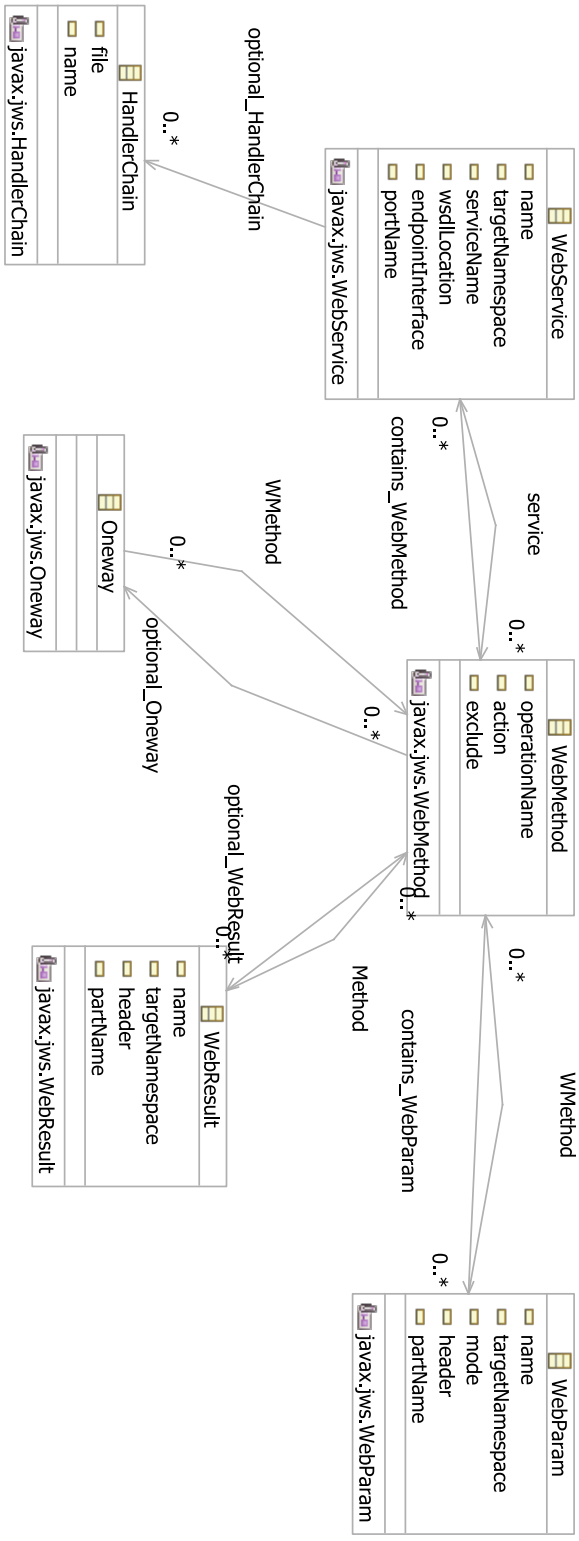


Figure 6.3: JWS annotation model

AssociationOverride	AssociationOverrides	AttributeOverride	AttributeOverrides
Basic	Column	ColumnResult	DiscriminatorColumn
DiscriminatorValue	Embeddable	Embedded	EmbeddedId
Entity	EntityListeners	EntityResult	Enumerated
ExcludeDefaultListeners	ExcludeSuperclassListeners	FieldResult	GeneratedValue
Id	IdClass	Inheritance	JoinColumn
JoinColumns	JoinTable	Lob	ManyToMany
ManyToOne	MapKey	MappedSuperclass	NamedNativeQueries
NamedNativeQuery	NamedQueries	NamedQuery	OneToMany
OneToOne	OrderBy	PersistenceContext	PersistenceContexts
PersistenceProperty	PersistenceUnit	PersistenceUnits	PostLoad
PostPersist	PostRemove	PostUpdate	PrePersist
PreRemove	PreUpdate	PrimaryKeyJoinColumn	PrimaryKeyJoinColumns
QueryHint	SecondaryTable	SecondaryTables	SequenceGenerator
SqlResultSetMapping	SqlResultSetMappings	Table	TableGenerator
Temporal	Transient	UniqueConstraint	Version

Table 6.5: Annotation types for the JPA

check properties that `Oneway` induce on the `WebMethod` the inverse association is needed (from `Oneway` to `WebMethod`).

6.2.5 Evaluation

In addition to the advantages of the use of ModelAn and AVal annotations on the JWS annotation framework cited for the Fraclet case (see section 6.1.5); an additional advantage comes from the use of the `Targets` annotation to represent implicit annotated elements. In this case, the specification states that methods belonging to a `WebService` class are by default `WebMethodS`. Through the `Targets` annotation, the annotation framework developer can capture this knowledge, and seamlessly implement this behavior thanks to the targeting processor that is part of the ModelAn tool chain. Using both AVal and ModelAn annotations we are able to implement all the constraints defined in the JWS specification.

The ModelAn model extraction and instantiation generation engine generate 2265 lines of code, 236 of them for the instantiation processor.

6.3 Java Persistence API

The Java Persistence API consists of the set of annotations defined in the EJB3 specification [MK06] that deal with the Object-Relational mapping for entities. The JPA defines 64 annotations (presented in table 6.5) that are used by application developers to specify how their entities will be persisted in a database. The specification distinguishes between annotations that define entities and their corresponding entity managers, from annotations that give the Object/Relational mapping.

The constraints to which an annotated program must adhere to are not well defined in the specification, and the relationships between the 64 annotations quite complex, further emphasizing the need for explicit constraints and modeling such as the ones provided by AVal and ModelAn.

```
1 @Entity
2 public class Customer { @Id
3     @GeneratedValue(strategy=AUTO) Long id;
4
5     @Version protected int version;
6
7     @ManyToOne Address address;
8
9     @Basic String description;
10
11    @OneToMany(targetEntity=com.acme.Order.class, mappedBy="customer")
12     Collection orders = new Vector();
13
14    @ManyToMany(mappedBy="customers")
15    Set<DeliveryService> serviceOptions = new HashSet();

17    public Long getId() { return id; }

19    public Address getAddress() { return address; }

21    public void setAddress(Address addr) { this.address = addr; }
22 }
```

Figure 6.4: Example: JPA annotated class

6.3.1 Description

In order to give an idea of the way in which AVal and ModelAn are used to specify and model the JPA we apply it to a subset of the JPA annotations. The rest of the annotations, their constraints and associations can be found on ModelAn's web page[18]. The selected subset consists of the annotations used to define entities and their mappings, to define overrides to those mappings, to define associations between entities, and to define tables and columns. The selected annotations are summarized in table 6.6

6.3.2 Example application

To give an idea of how the JPA is used to specify the persistence of an entity, we present the `Customer` entity implemented in figure 6.4. The example provided is taken from the JPA specification.

The `Customer` class is marked as an entity, as specified by the `Entity` on line 1. By marking the class `Entity`, the interpretation engine will annotate it with `Table` as well, using the default values for its elements. The same strategy is applied for each of the fields/properties of the class with the `Column` annotation. This clears the source code from unnecessary annotations, but introduces hidden annotations that may hinder the understanding of the source code.

The fields of the class are annotated with a number of JPA annotations that describe how to persist it: The `id` field serves as id for the table which is assigned a generated value (line 3). The version field will keep a value that is used as optimistic lock value (line 5). The `Basic` annotation in the `description` field on line 9 states that the value of the field is to be mapped to a basic field on the primary table of the entity.

[18] http://spoon.gforge.inria.fr/ModelAn

Annotation	**Location**	**Parameter**	**Description**
`Entity`	Class	*name*	Specifies a class as an entity.
`MappedSuperclass`	Class	-	Designates a class whose mapping is to be applied to entities that inherit from it.
`AttributeOverride`	Class,Method,Field	*name, columns*	Overrides the mapping of a property or field defined in a mapped superclass.
`AssociationOverride`	Class,Method,Field	*name,joinColumns*	Overrides a many-to-one or one-to-one property or field relationship defined in a mapped superclass.
`ManyToMany`	Method,Field	*targetEntity, cascade, fetch, mappedBy*	Defines a many-valued association with a many-to-many multiplicity with the *targetEntity*.
`OneToOne`	Method,Field	*targetEntity, cascade, fetch, optional, mappedBy*	Defines a single-valued association to a *targetEntity*.
`ManyToOne`	Method,Field	*targetEntity, cascade, fetch, optional*	Defines a single-valued association to a *targetEntity*.
`OneToMany`	Method,Field	*targetEntity, cascade, fetch, mappedBy*	Defines a many-valued association to a *targetEntity* with a many-to-one multiplicity.
`Table`	Class	*name, catalog, schema, uniqueConstraints*	Defines the primary table for an entity.
`Column`	Method,Field	*name, unique, nullable, insertable, updatable, columnDefinition, table, lenght, precision, scale*	Specifies a mapped column for a persistent property or field.

Table 6.6: Selected JPA annotation description

Fields of the class are also used to state relationships of the entity to other entities. The `address` field represents a `ManyToOne` relation (line 7 with the `Address` entity. The `orders` field, contains a `OneToMany` relation (line 12)with the `Order` class; note that since just by looking at the type of the field it is not possible to know the type of the elements of the collection, the `targetEntity` element of the `OneToMany` annotation is used. Also, the `mappedBy` element states that the inverse relation on the `Order` entity is specified by the field called `customers`. The `serviceOptions` field contains a `ManyToMany` relation (line 14) to the `DeliveryService` entity. As with the `orders` relation, this one is mappedBy a field called `customers` however, in contrast with the `orders` relation, the `targetEntity` is not specifies, since type of the objects contained in the `serviceOptions` field can be inferred from its type parameter.

6.3.3 Constraints

We summarize the constraints for the selected JPA annotations in table 6.7. The constraints are derived from the framework's specification. The specification, in contrast with JWS's, does not always explicitly states constraints on the use of the annotations, so some of the constraints here defined stem from our interpretation of the wording of the spec.

To give an example of such interpretation, take the `Entity`'s constraint that states that the annotation cannot be placed on a class that has the `MappedSuperclass` annotation. This constraint is not explicitly stated in the specification, but given that mapped superclasses have no associated table and that the annotation of a class with `Entity` implies an annotation of `Table` with the default parameters, the use of `Entity` forbids the use of `MappedSuperclass`.

6.3.4 Annotation model

Having defined the constraints for the selected annotations of the JPA, we now present how ModelAn and AVal annotations are used to define JPA's annotation model, and its corresponding constraints.

Entity

```
1 @Retention(RUNTIME)

3 @AValTarget(CtClass.class)
4 @Prohibits(MappedSuperclass.class)
5 public @interface Entity {

7  @Default("self.target_CLASS.SimpleName")
8  String name() default "";
9 }
```

The `AValTarget` annotation is used (line 3) instead of Java's `Target` annotation to narrow the target of the annotation to only classes, since `Target` can only define as target a type. The `Prohibits` in line 4 is used to forbid the use of `Entity` on classes already annotated with

`Entity`	• Must only be placed on Classes • The class cannot be annotated with `MappedSuperclass`
`MappedSuperclass`	• Must only be placed on Classes • The class cannot be annotated with `Entity`
`AttributeOverride`	• If it is place on a class, the class must extend a type annotated with `MappedSuperclass` • If on a field or method, the type of the field/method must be annotated `MappedSuperclass` • The `name` element must be a `Column` of the overridden type
`AssociationOverride`	• If it is place on a class, the class must extend a type annotated with `MappedSuperclass` • If on a field or method, the type of the field/method must be annotated `MappedSuperclass` • The `name` element must be a relation (`ManyToMany` or `OneToOne`) of the overridden type.
`ManyToMany`	• Type Collection • if collection is not generic,`targetEntity` element must be defined • The `targetEntity` element must refer to a class that is annotated with `Entity`
`OneToMany`	• Type is collection • if collection is not generic, OneToMany.targetType must be defined • The `targetEntity` element must refer to a class that is annotated with `Entity`
`OneToOne`	• The `targetEntity` element must refer to a class that is annotated with `Entity`
`ManyToOne`	• The `targetEntity` element must refer to a class that is annotated with `Entity`
`Table`	• Must be in classes already annotated with `Entity` • It cannot be in a class annotated with `MappedSuperclass`
`Column`	• Must be placed on fields/methods of classes annotated with `Table`

Table 6.7: Constraints for selected JPA annotations

`MappedSuperclass`. In accordance with the specification, the default value for the `name` element is made to be the simple name of the class on which the annotation is placed using the `Default` annotation (line 7).

Associations between `Entity` and other annotations are defined by AVal annotations on other annotation types.

MappedSuperclass

```
1 @AValTarget(CtClass.class)
2 @Prohibits(Entity.class)
3 public @interface MappedSuperclass {
4 }
```

Two AVal annotations are used to constraint `MappedSuperclass` annotations to classes (line 1) not already annotated with `Entity` (line 2).

AttributeOverride

```
1 @Target({TYPE, METHOD, FIELD})
2 @Retention(RUNTIME)

4 @RequiresAny({
5         @Requires(Entity.class),
6         @Requires(Embedded.class)
7 })

9 @Prohibits(MappedSuperclass.class)
10
11 @OCLConstraints({
12         @OCLConstraint("self.target_TYPE <> null implies"+
13 " MappedSuperclass.allInstances()->exists(m|m.target_TYPE.Reference"+
14                                 ".isAssignableFrom(self.target_TYPE.Reference))"),
15         @OCLConstraint("self.target_FIELD <> null implies"+
16 " MappedSuperclass.allInstances()->exists(m|m.target_TYPE.Reference"+
17                                 ".isAssignableFrom(self.target_FIELD.Type))"),
18         @OCLConstraint("self.target_METHOD <> null implies"+
19 " MappedSuperclass.allInstances()->exists(m|m.target_TYPE.Reference"+
20                                 ".isAssignableFrom(self.target_METHOD.Type))"),

22         @OCLConstraint("self.target_TYPE <> null implies"+
23 " Column.allInstances()->exists(c|c.name = self.name and "+
24     " c.Parent.oclAsType(CtType).Reference.isAssignableFrom(self.target_TYPE.Reference)"),
25         @OCLConstraint("self.target_FIELD <> null implies"+
26 " Column.allInstances()->exists(c|c.name = self.name and"+
27     " c.Parent.oclAsType(CtType).Reference.isAssignableFrom(self.target_FIELD.Type)"),
28         @OCLConstraint("self.target_METHOD <> null implies "+
29 "Column.allInstances()->exists(c|c.name = self.name and"+
30     " c.Parent.oclAsType(CtType).Reference.isAssignableFrom(self.target_METHOD.Type)")
31 })
32 public @interface AttributeOverride {
33     String name();
34     Column column();
35 }
```

`AttributeOverride` annotations require the type on which they are placed to be either an entity or an embedded class (line 4). Since attribute overrides are used to change the mappings defined on the class, the class cannot be annotated with `MappedSuperclass` (line 9).

Finally, two constraints are implemented in several OCL expressions. The constraint that states that if the annotation is placed on a class, the class must extend a type annotated with `MappedSuperclass` and if placed on a field or method, the type of the field or return type of the method must be annotated with `MappedSuperclass` are specified by the expressions on lines 12, 15 and 18. All of these expressions first check if the corresponding target (class, field or method) is present, and then traverse the instances of `MappedSuperclass` to check if one of them is placed on a type which is a super type of the current target.

The second constraint states that the `name` of the overridden attribute must be a name of a column defined on a super class. For this, again three expressions (lines 22, 25 and 28) are used. As with the previous constraints, first we check the target on which the `AttributeOverride` annotation is placed. For each target, all the `Column` instances are traversed, and if they are placed on a type which is a super type of the current type, there must exist one whose name is equal to the name defined on the current `AttributeOverride` annotation.

AssociationOverride

```
1 @Target({TYPE, METHOD, FIELD})
2 @Retention(RUNTIME)

4 @Associations({
5   @Association(name="overriddenAssociationM2O",type=ManyToOne.class,
6 query="ManyToOne.allInstances()->select(m2o | m2o.column.name = self.name )"),
7   @Association(name="overriddenAssociationO2O",type=OneToOne.class,
8 query="OneToOne.allInstances()->select(o2o | o2o.column.name = self.name)"),
9   @Association(name="entity",type=Entity.class,
10 query="Entity.allInstances()->select( e | e.target_CLASS = self.target_TYPE "
11                                   +"or self._annotation.Parent = e.target_CLASS)")
12 })

14 @OCLConstraints({
15   @OCLConstraint("self.target_TYPE <> null implies MappedSuperclass.allInstances()->"+
16 "exists(m|m.target_TYPE.Reference.isAssignableFrom(self.target_TYPE.Reference))"),
17   @OCLConstraint("self.target_FIELD <> null implies MappedSuperclass.allInstances()->"+
18 "exists(m|m.target_TYPE.Reference.isAssignableFrom(self.target_FIELD.Type))"),
19   @OCLConstraint("self.target_METHOD <> null implies MappedSuperclass.allInstances()->"+
20 "exists(m|m.target_TYPE.Reference.isAssignableFrom(self.target_METHOD.Type))"),

22   @OCLConstraint("self.overriddenAssociationM2O->notEmpty() "
23               +"or self.overriddenAssociationO2O->notEmpty()")
24 })

26 public @interface AssociationOverride {
27     String name();

29     JoinColumn[] joinColumns();
30 }
```

The `AssociationOverride` annotation type has a relationship for each of the associations it overrides, as defined by the `Association` annotations in lines 5 and 7. It also defines a relation to the entity for whom the associations are overridden (line 9).

In terms of constraints, `AssociationOverride`, has constraints similar to `AttributeOverride`. If placed on a type, the type must extends a type annotated with `MappedSuperclass` (line 15). If placed on a field or method, the type of the field, or return type of the method, must be annotated with `MappedSuperclass` (lines 17 and 19). Finally, the overridden associations must exist, that is, either the `overriddenAssociationM2O` or `overriddenAssociationO2O` re-

lations must not be empty (line 22).

ManyToMany

```
1 @Target({METHOD, FIELD})
2 @Retention(RUNTIME)

4 @Inside(Entity.class)
5 @Associations({
6   @Association(name="assocTarget",type=Entity.class,
7                query="Entity.allInstances()->"+
8               "select(e|e.target_CLASS.QualifiedName = self.targetEntity.QualifiedName)"),
9   @Association(name="otherSide",type=ManyToMany.class,
10 query="ManyToMany.allInstances()->select(m2m|m2m.mappedBy = self.column.name"),
11   @Association(name="column",type=Column.class,
12 query="Column.allInstances()->select(c| c._annotation.Parent = self._annotation.Parent")
13 })

15 @Type(Collection.class)
16 @OCLConstraints({
17  @OCLConstraint("self.target_METHOD <> null implies "
18 +"self.target_METHOD.Type.ActualTypeArguments.isEmpty() implies self.targetEntity <> null"),
19   @OCLConstraint("self.target_FIELD <> null implies "
20 +" self.target_FIELD.Type.ActualTypeArguments.isEmpty() implies self.targetEntity <> null"),

22   @OCLConstraint("self.assocTarget->notEmpty()")
23 })
24 public @interface ManyToMany {
25 @Default(
26 "if self._annotation.Parent.oclAsType(CtTypedElement).Type.ActualTypeArguments->isEmpty()"+
27   " then null "+
28 " else self._annotation.Parent.oclAsType(CtTypedElement).Type.ActualTypeArguments.first()")
29   Class targetEntity() default void.class;
30   CascadeType[] cascade() default {};
31   FetchType fetch() default LAZY;
32   String mappedBy() default "";
33 }
```

The `ManyToMany` annotation represents relationships between two entities (as described by the `Inside` and `Association` annotations in lines 4 and 6). A `ManyToMany` relation declared by an entity can have a corresponding `ManyToMany` inverse relation on the target entity (`Association` annotation on line 9). Also, for convenience, an association between the `ManyToMany` relation and the column that represents it on the database is defined in line 11.

In terms of constraints, the type of the field on which the `ManyToMany` annotation is placed should be a Collection, which is checked with the `Type` AVal annotation in line 15. In lines 17 and 19, an OCL expression is used to validate that if the collection type of the field or property that represents the many to many relation has no type-parameters, then the `targetEntity` element must be defined. This constraint is closely related with the default value expressed in line 23, which says that the value for the `targetEntity` annotation is the generic type of the collection or null if no type parameter is used. For an example of this, see the listing in section 6.3.2.

OneToOne

```
1 @Target({METHOD, FIELD})
2 @Retention(RUNTIME)
```

```
4 @Inside(Entity.class)
5 @Associations({
6   @Association(name="assocTarget",type=Entity.class,
7 query="Entity.allInstances()->"
8      +"select(e|e.target_CLASS.QualifiedName = self.targetEntity.QualifiedName)")
9   @Association(name="column",type=Column.class,
10 query="Column.allInstances()->select(c| c._annotation.Parent = self._annotation.Parent")

12 })
13 @OCLConstraint("self.assocTarget->notEmpty()")
14 public @interface OneToOne {

16  @Default("self._annotation.Parent.oclAsType(CtTypedElement).Type")
17  Class targetEntity() default void.class;

19  CascadeType[] cascade() default {};
20  FetchType fetch() default EAGER;
21  boolean optional() default true;
22  String mappedBy() default "";
23 }
```

The `OneToOne` annotation defines relations between entities (represented with the `Inside` and `Association` annotations in lines 4 and 6). It also defines a convenience association with the column to which the relation is mapped in the entities table, line 9 . Finally, the value for the `targetEntity` defaults to the type of the field or method on which the `OneToOne` annotation is placed (line 16). This type must be annotated with `Entity`, that is, the `assocTarget` association must not be empty (line 13).

ManyToOne

```
1 @Target({METHOD, FIELD})
2 @Retention(RUNTIME)

4 @Inside(Entity.class)
5 @Associations({
6   @Association(name="assocTarget",type=Entity.class,
7 query="Entity.allInstances()->"
8       +"select(e|e.target_CLASS.QualifiedName = self.targetEntity.QualifiedName)")
9   @Association(name="column",type=Column.class,
10 query="Column.allInstances()->select(c| c._annotation.Parent = self._annotation.Parent")

12 })
13 @OCLConstraint("self.assocTarget->notEmpty()")
14 public @interface ManyToOne {
15  @Default("self._annotation.Parent.oclAsType(CtTypedElement).Type")
16  Class targetEntity() default void.class;
17  CascadeType[] cascade() default {};
18  FetchType fetch() default EAGER;
19  boolean optional() default true;
20 }
```

As with the two previous annotations, `Inside` and `Association` are used in lines 4 and 6 to define the relation between entities, and an additional `Association` is used to tie the `ManyToOne` annotation to its corresponding `Column` (line 9). As with the previous annotation, the type of the field/method gives the default value for the `targetEntity` element (line 15). Also, the `assocTarget` association must not be empty (line 13).

OneToMany

```
1 @Target({METHOD, FIELD})
2 @Retention(RUNTIME)

4 @Inside(Entity.class)
5 @Associations({
6         @Association(name="assocTarget",type=Entity.class,
7 query="Entity.allInstances()->"+
8       "select(e|e.target_CLASS.QualifiedName=self.targetEntity.QualifiedName)")
9   @Association(name="column",type=Column.class,
10 query="Column.allInstances()->select(c| c._annotation.Parent = self._annotation.Parent")

12 })

14 @Type(Collection.class)
15 @OCLConstraints({
16  @OCLConstraint("self.target_METHOD <> null implies"+
17  " self.target_METHOD.Type.ActualTypeArguments.isEmpty() implies self.targetEntity <> null"),
18  @OCLConstraint("self.target_FIELD <> null implies"+
19  " self.target_FIELD.Type.ActualTypeArguments.isEmpty() implies self.targetEntity <> null"),
20 @OCLConstraint("self.assocTarget->notEmpty()")
21 })
22 public @interface OneToMany {
23 @Default("if self._annotation.Parent.oclAsType(CtTypedElement).Type.ActualTypeArguments->isEmpty()"+
24   " then null "+
25   " else self._annotation.Parent.oclAsType(CtTypedElement).Type.ActualTypeArguments.first()")
26  Class targetEntity() default void.class;

28  CascadeType[] cascade() default {};
29  FetchType fetch() default LAZY;
30  String mappedBy() default "";
31 }
```

The associations and constraints for the `OneToMany` annotation are analogous to those for the `ManyToMany`: the `assocTarget` (line 6) to the associated entity and `column` (line 9) to the column to which the relationship is mapped, the `Inside` AVal annotation (line 4) to specify the owner of the annotation. Also, the `OneToMany` annotation type is meant to be placed on fields/methods of type Collection (line 14), and if the collection has no type parameter the `targetEntity` element must be defined and point to a type annotated with `Entity` (lines 23 and 20).

Table

```
1 @Target(TYPE)
2 @Retention(RUNTIME)

4 @Requires(Entity.class)
5 @Prohibits(MappedSuperclass.class)
6 @Association(name="entity",type=Entity.class,
7    query="Entity.allInstances->select(e|e.target_CLASS = self.target_TYPE)")
8
9 @Targets("CtClass.allInstances()->select(c|c.Annotations->"
10                +"exist(a|a.AnnotationType.SimpleName = 'Entity')")
11 public @interface Table {

13  @Default("self.entity.name")
14  String name() default "";

16  String catalog() default "";
17  String schema() default "";
```

```
18 UniqueConstraint[] uniqueConstraints() default {};
19}
```

Given that by default all entities are persisted, the `Table` annotation uses the `Targets` annotation to explicit the mapping of the `Entity` classes to a database table. For this, an OCL expression (line 9) traverses all the classes in the application, looking for those that have an `Entity` annotation and annotates them with `Table` if they are not already annotated.

Since `MappedSuperclass`es do not have an associated table, classes annotated with it cannot be annotated with `Table` as specified by the `Prohibits` annotation in line 5. The `name` of the table defaults to the name of the corresponding `Entity` (line 13). To express this, a convenience association called `entity` is defined in line 7.

Column

```
1 @Target({METHOD, FIELD})
2 @Retention(RUNTIME)


5 @Inside(Table.class)
6
7 @Targets("CtField.allInstances()->select(f|" +
8   "not f.Annotations->exists(a|a.AnnotationType.SimpleName = 'Transient') and" +
9   "f.Parent.Annotations->exists(a|a.AnnotationType.SimpleName = 'Entity'"+
10                              " or a.AnnotationType.SimpleName = 'MappedSuperclass)"
11                "))")

13 @Association(name='belongs_table',type=Table.class,
14  query="Table.allInstances->select(t|t.contains_Column = self).first()")

16 public @interface Column {

18   @Default("if self.target_FIELD <> null then self.target_FIELD.SimpleName " +
19         "else " +
20  "self.target_METHOD.SimpleName.substring(3,self.target_METHOD.SimpleName.size())")
21  String name() default "";

23  boolean unique() default false;
24  boolean nullable() default true;
25  boolean insertable() default true;
26  boolean updatable() default true;

28  String columnDefinition() default "";

30  @Default("self.belongs_table.name")
31  String table() default "";

33  int length() default 255;
34  int precision() default 0;
35  int scale() default 0;
36 }
```

The `Column` annotation is to be placed in methods or fields that belong to a class annotated with `Table` (line 5). By default, all fields of classes annotated with `Entity` or `MappedSuperclass` carry an implicit `Column` annotation unless they are annotated with `Transient`. This is specified by the `Targets` ModelAn annotation in line 7. A link to the table to which each column belongs is defined by the `Association` in line 13. Using this association, the default value for the `table` element is defined (line 30). The default `name` of the column is extracted from the name of the field or method on which the annotation is placed

(line 18).

By interpreting the ModelAn and AVal annotations present in the selected JPA annotation types, we extracted the annotation model shown in figure 6.5.

6.3.5 Evaluation

Using ModelAn and AVal annotations we were able to explicitly state most of the constraints, associations and implicit annotations defined by the JPA specification. Nevertheless, the large number JPA annotation types, and the complexity of the relations between them put a strain in the use of ModelAn and AVal annotations to describe both constraints and associations. In places, the number of lines of code devoted to meta-annotations supersede the lines of code for the definition of the annotation type itself. For example, most of the complexity on the associations stem from the relations of different annotation types to the `Entity` and `MappedSuperclass` annotations. In the abstract, `MappedSuperclass` represents the same concept as `Entity`, since `MappedSuperclass` defines the mapping of its sub-classes (sub-entities) to the database. Because of this, all the annotations in JPA that serve to specify persistence (almost all of them) are related to both `Entity` and `MappedSuperclass`. To us, this means that `Entity` and `MappedSuperclass` are actually sub-annotations[19] of an abstract annotation, call it `Mappable`. If such a super annotation existed, then all the mapping-defining annotations (`Table`, `NamedQuery`, etc) would be linked to it, instead of to both `Entity` and `MappedSuperclass`; cutting the number of associations in the model by roughly half. However, since inheritance between annotation is forbidden by the Java language, the annotation model becomes much more complex.

Another factor that makes the definition of JPA's annotation model harder is the reliance of the specification on *properties*. Properties are a pair of methods that encapsulate the access to a field (getters and setters). The JPA uses fields and properties in an interchangeable manner in many places (for example, the `Column` annotation can be placed on a field or a method *that represents a property*). This makes the definition of constraints for annotations that can go on either places more verbose since the constraint must be replicated in the case that the annotation is placed on a method or on a field.

The generated Java classes that reify the 64 annotations of the JPA amount to a total of over twelve thousand lines of code, eight hundred of them dedicated to the source code processor that instantiates the annotation model from an annotated application.

6.4 Summary

In this chapter we have shown how ModelAn and AVal can be used to specify the annotation model and corresponding consistency constraints in three industrial annotation frameworks of different size.

[19] Here sub-annotation is taken similarly to subclass

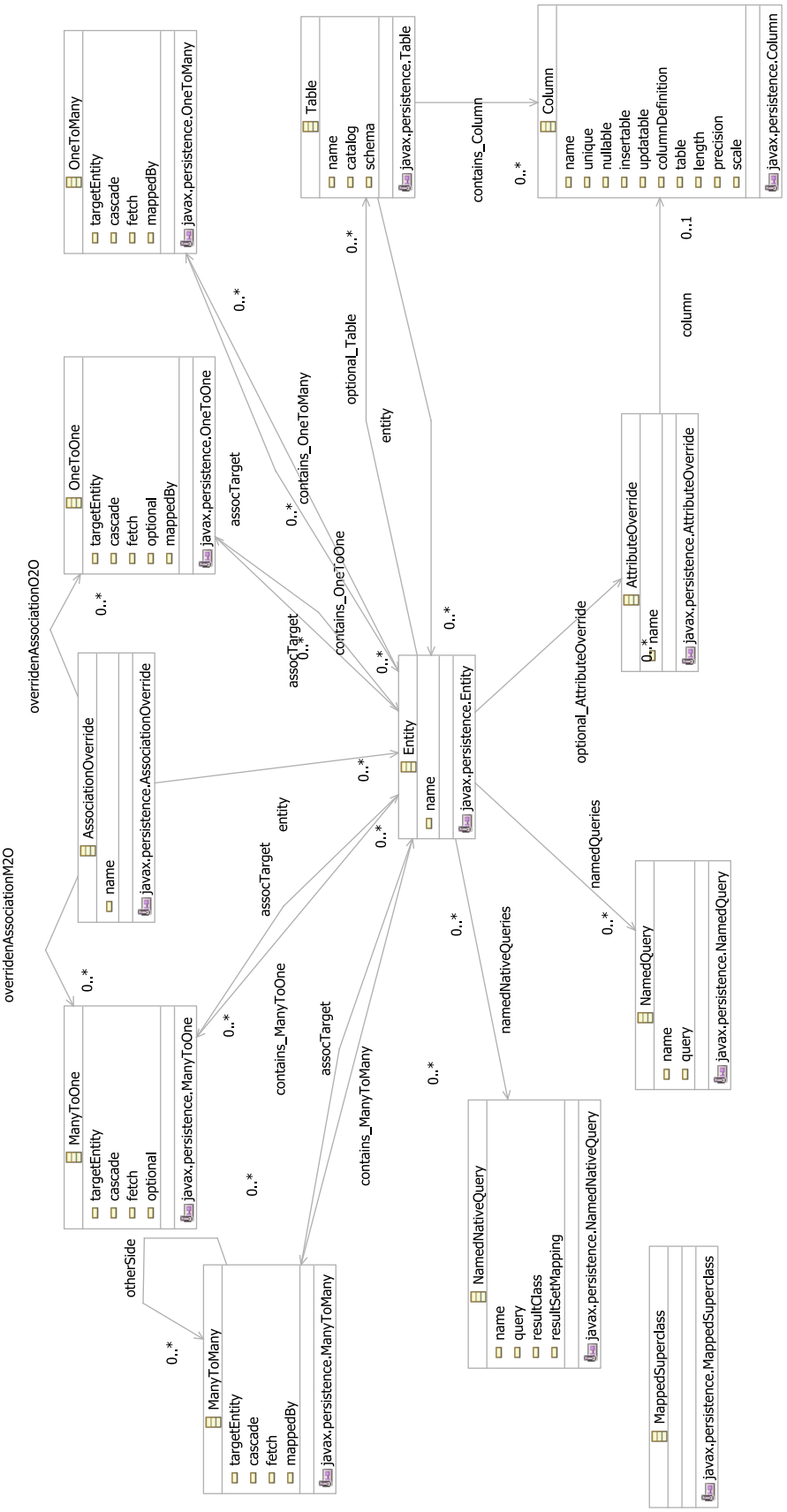


Figure 6.5: Subset of the JPA's annotation model

Fraclet is an annotation framework for the Fractal component model. With Fraclet's annotations, the developer defines the mapping of a class that serves as implementation of a component to the architecture of the application defined in the ADL. We use ModelAn and AVal to define the constraints identified for Fraclet, and to extract its corresponding annotation model (section 6.1).

JWS is an annotation framework defined by Java for the development of WebServices. Again, we use ModelAn and AVal to construct the annotation model corresponding to the framework, and using this model we are able to validate all the constraints define by the specification. In particular, we are able to describe *implicit annotations* on elements of the program. In the case of JWS, by default all methods belonging to a class that represent a web service are treated as if they were annotated with `WebMethod`. We are able to reproduce this behavior by employing the `Targets` annotation described in section 5.3.1.0.

JPA The Java Persistence Framework describes sixty-four annotations to describe how entities in EJB3 are persisted. This is the largest of the three case studies, and in this document we present in detail ten annotations. The annotation model that resulted from applying ModelAn annotations to the reference implementation of JPA describes the associations between all sixty-four annotations, which reduces the effort in understanding the framework. In addition to this, we extracted the constraints implicit in the specification of JPA, and translated them into AVal and OCL constraints. Nevertheless, the complex nature of the annotation types defined in JPA show some of the limitations of both annotations as an abstraction mechanism and the use of ModelAn and AVal to specify them. First, we found that the number of meta-annotations required to fully specify a JPA annotation-type would sometimes exceed the number of lines of code needed to define the annotation type. This is a testament of the importance of explicitly stating the semantics of annotation types. Second, the constraints we identified remain one possible interpretation of the specification, since in contrast with JWS's, JPA's specification does not explicitly address the constraints on the use of the annotations it defines. Finally, some annotation types defined in JPA hint at the limitations of Java annotations to define complex concepts: lack of inheritance begets repetition of constraints and associations; and since annotations cannot be repeated on a place in the code, almost all annotation types are defined with an accompanying collection annotation type, which nearly doubles the number of annotation types.

A summary of the case studies presented in this chapter can be seen in Table 6.8. It is interesting to note that, although the three case studies define a similar number of annotation types (between 7 and 10), their complexity is quite different. The number of constraints for the JPA is more than double than for Fraclet, although the former has only four more annotation types than the later. The use of AVal and ModelAn meta-annotations exposes this otherwise hidden complexity.

The use of ModelAn and AVal on the three annotation frameworks evidences some strengths and weaknesses of our approach. On one hand, the annotation model gives an insight on the way on which different annotation types relate to each other, in the complete

Framework	Annotation types	Constraints	Default Values	Targets	Generated Code (SKLoC)
Fraclet	6	14	2	0	1.7
JWS	7	25	4	1	2
JPA	10 (of 64)	42	8	2	12

Table 6.8: Summary of the case studies

model, almost all of the annotations have some sort or relation with other annotations; which, by looking at the un-annotated source code is not evident. Also, we were able to specify the behavior of implicit annotations, such as the tables that accompany entities, or the columns that accompany fields and methods of entity classes; these are also not necessarily evident in the source code of the application.

On the other hand, the amount of ModelAn and AVal annotations may overwhelm the programmer, in particular in annotation types for which they total more lines of code than the annotation type itself. In these cases, it would be worthwhile to search for abstraction mechanisms to be able to factorize common OCL expressions for example. This can be achieved by an OCL header annotation at package level that would contain `let` expressions that would be available to all `OCLConstraint` and `Association` annotations present in code elements with in the package.

In the next chapter, we will present perspectives and conclude.

Part IV

Conclusions and Future Work

7

Conclusion and Perspectives

Contents

This chapter presents a summary of the contributions of this thesis, and gives perspectives on future work.

7.1 Contributions

The objective of this thesis is to provide tools and techniques to aid in the development of annotation frameworks. In order to do this, we analyzed the design and development of annotation frameworks and identified four challenges that the annotation framework developer must deal with:

I The representation of domain concepts as annotation types

II The mapping of annotation types to code elements

III The definition of constraints to validate annotated programs

IV The reification of annotations for their interpretation.

We have achieved this in two parts: First by introducing generic constraints to specify annotation frameworks and an annotation framework that uses generic constraints to embed consistency constraints in the source code of the annotation framework and latter checks the constraint in an annotated program (challenge `III`). Second, by defining annotation models that augment annotation types with the notion of association between annotation types, we rise the level of abstraction of annotation types closer to that of their domain models, making their representation easier (challenge `I`). Using this annotation model, the annotation framework developer can explicitly define the relation between annotation types and the code on which they are placed (challenge `II`). Finally, the annotation model is used to generate classes that represent the annotations present on an application, thus addressing challenge `IV`.

General contributions of the approaches are described below.

7.1.1 General Contributions

Exposing Complexity The semantics of an annotation are hidden in its interpretation, while its constraints are informally described in its documentation. Looking at the case studies, there is a stark contrast in complexity between Fraclet and JPA. In Fraclet we have identified 14 constraints for 6 annotation types (giving an average of 2.3 constraints per annotation); while in the JPA we identified 42 constraints for the 10 annotation types analysed (4.2 constraints per annotation). This suggests that the number of annotation types that a framework defines is not a good indicator of its complexity.

By using AVal/ModelAn to specify the constraints, default values and targets of the annotation types of a framework, its complexity is exposed. This will make the use of the annotation framework easier, since the annotation user will know the constraints that it must respect in order to develop annotation-wise valid programs.

Enhanced Expressiveness The annotations defined by AVal and ModelAn enhance the expressiveness of the Java language when defining new annotation types. The `targets` annotation extends the Java-provided `target` by allowing the developer to specify, not only to which kinds of code elements an annotation can be bound to; but also to which elements in particular. This permits the developer to express *implicit* annotations found in the Java Web Services and Java Persistence API case studies c.f. 6.26.3.

The annotation framework developer, through AVal annotations and OCL constraints, can express the constraints to which programs that use his annotations must adhere to. The use of AVal does not only allows the definiton of constraints, but it also provides a way to check them, thereby reducing the development type of annotation frameworks and enhancing the interpretation engine's cohesion by removing the validation related concerns from its code.

Enhanced Code Comprehension Applications developed with annotation frameworks that use AVal-ModelAn are easier to comprehend. The visualization of the annotation model that is represented by the annotations in the program presents an additional

domain-specific view of the application. In this view, the developer navigates concepts of the domain, rather than Java concepts which are mapped to concepts of the domain.

The use of AVal-ModelAn also enhances the understandability of annotation frameworks that use them. First, the annotation model exposes the relations between annotation types, and the default value and targets annotation make explicit the semantics of annotations and their elements.

Contributions specific to each of the two parts of the approach are desribed below.

7.1.2 Generic constraints

We have developed a classification of the constraints present in annotation frameworks depending on the subject of the constraint. Two classes of constraints are defined: **annotation-wise** for constraints that deal with the properties of annotations and their elements; and **code-wise** for constraints that deal with the properties of the code on which annotations are placed. By analyzing existing annotation frameworks we identified seven generic constraints; and based on them, implemented the AVal annotation framework. AVal counts with the following properties:

Generic AVal annotations, as demonstrated in chapter 6, can be applied to real annotation frameworks, and common constraints can be expressed with a (combination of) AVal annotations. In addition to this, generic constraints can be used to extract the annotation model of the annotation types that they constraint. For example, if an annotation requires another one, this can be seen as an optional association between them.

However, since generic constraints must remain generic, they cannot address constraints specific to a given annotation framework. In these cases, the extensibility of AVal's implementation allows for easily implementing new constraints.

Declarative AVal constraint annotations are a declarative specification for the annotation types in a framework. The framework developer does not need to explicitly state the manner in which the constraints will be checked in annotated programs.

Parameterizable Most constraint annotations in AVal can be parametrized in order to customize the messages presented to the user when constraints are violated. In addition to this, it is possible to attach a code transformation to each AVal annotation that will provide a way to fix the error. The possibility of customize the error messages is essential to AVal, given that, since its constraints are generic, their default error messages are equally generic and give little insight to the user on the cause of the error or what steps to take in order to fix it. The knowledge of both lays with the annotation framework developer.

Extensible AVal provides an API that allows annotation framework developers to construct their own constraint annotations when deemed necessary.

Composable In chapter 4.4 we outline the way in which constraint annotations are composed while respecting the constraints of the Java language on the placement of annotations on a code element.

Enhances code understandability The inclusion of AVal constraints on annotation types enhances their understandability, since the rules of use of the annotations are explicitly stated. This facilitates their use by application developers.

AVal's generic constraints, when compared to other approaches in literature (section 2.6.4) is the only one to allow for both annotation and code-wise constraints. In addition to this, it is the only one to provide an extension mechanism to define new constraints, and customization points for defining error messages and quick-fix transformations.

7.1.3 Annotation models

We propose an extension to annotation types in Java to express relations between them. Using this extension, we address the three remaining challenges, while complementing challenge `III` addressed by AVal. Challenge `I`, the representation of domain concepts as annotations is addressed by constructing an annotation model from the annotation types and the relations between them. The mapping between annotation types and code elements, challenge `II`, is addressed by augmenting annotation types with OCL queries that resolve to the elements to which they are mapped. We also extend AVal, so that constraints can be defined as OCL expressions on this annotation model. Finally, the elements defined in the annotation model are transformed to Java classes that serve as reified annotations present in the application. These reified annotations can be used to interpret the application's source code (challenge `IV`).

The extension is realized through the `Association` annotation that is offered by the ModelAn annotation framework that we have developed. Using this extensions, we extract an *annotation model* that represents the annotation types, their relations, the Java AST and the relations between the annotation types and the language's AST. The construction of the annotation model carried out by ModelAn takes the annotation types annotated with `Association` and constructs an Ecore-based model that represents the annotation types, their attributes, their relation to Java's AST and the relations defined by the `Association` annotation. In parallel to the construction of this model, a source-code processor that instantiates this model is generated. The model instantiation engine takes an annotated application and constructs an instance of the annotation model generated before. It is on this instance that constraints, default values and mapping information between the framework's annotation types are applied.

Annotation models count with the following properties:

Complements annotation type definition The addition of associations to annotation types brings them closer to the domain model that they represent. By defining associations, the framework developer can make the transition between the domain model and the development of annotation types easier.

High-level view of annotation framework The domain model also gives a higher level view on the annotation framework than the one provided by the source code alone. In it, interactions between annotation types are explicitly stated. This makes the annotation model an useful tool for documenting the framework.

Bridges annotations and models Annotation models also bridge the gap between annotations and models. The annotation framework developer can take advantage of tools and techniques existing in the model-driven engineering domain, and apply them to the annotation framework development. In this thesis we exploit this bridge by implementing annotation constraints and queries in the OCL, and by leveraging the tools offered by the Eclipse Modeling Framework to implement the reification of annotations.

Extensible Model Although an annotation model represents the annotation types of a single framework, several instances of different annotation model can co-exist in a same application. When an application uses different annotation frameworks, their annotation model instances are merged into a large model that includes representation for all annotations as well as for the code in the application.

Multidimensional View of an Application Since several annotation models co-exist in an application, each annotation model instance represents a domain-specific view. If a large application uses an annotation framework for persistence and another for web-page navigation; then there will be an annotation model instance that defines the application's navigation graph, and another that shows the ER persistence schema.

Constraint Checking We extend AVal's generic constraint annotations with an annotation that takes as parameter an OCL expression that will be evaluated in the annotation model's instance. The use of OCL to express constraints does not require knowledge of the Spoon API, and therefore should be more accessible to annotation framework developers.

Definition of Complex Default Values By using OCL queries over the annotation model, we allow the framework developer to describe complex default values for annotation elements. No other annotation development tool allows for this.

Explicit Annotation-Code Relations Also through OCL queries, we allow the annotation framework developer to specify the relation between annotations and the code on which they are placed. This makes annotations closer to aspect by providing them with a pointcut mechanism. It is also used to express implicit annotations present in complex annotation frameworks.

7.2 Comparison with other approaches

In order to position the work of this thesis with relation to others in the field, we compare AVal/ModelAn with the works presented in section 2.6 in tables 7.1, 7.2 and 7.3.

	Platform	**Representation**		
		Code	**Annotation**	**Support**
mTurnpike	Java	model	model	yes
XIRC	Java	XML	none	no[a]
ADC	.NET	DOM	none	yes
AVal/ModelAn	Java	AST[b]	model	yes

[a]Only represents annotations present in byte-code
[b]The AST is represented using EMF

Table 7.1: Annotated application representation in the compared approaches (including AVal/Modelan)

	Constraints			
	Code	**Annotation**	**Declarative/ Explicit**	**Embedded/ External**
mTurnpike	no	no	–	external
XIRC	yes	no[a]	explicit	external
ADC	no	yes	declarative	embedded
AVal/ModelAn	yes	yes	both[b]	both[c]

[a]No explicit support for annotation-wise constraints
[b]Declarative AVal constraints and explicit OCL constraints
[c]Meta-annotations and Dummy annotations

Table 7.2: Constraints offered by the compared approaches (including AVal/Modelan)

	Interpretation		
	Support	**Compile-time**	**Runtime**
mTurnpike	yes	transformations	–
XIRC	no	–	–
ADC	no	–	–
AVal/ModelAn	yes[a]	Transformations[b]	Partial[c]

[a]Although neither AVal nor ModelAn explicitly support interpretation, they are integrated into the Spoon annotation processing engine
[b]Using Spoon
[c]Annotation reification can be made available at runtime

Table 7.3: Interpretation support offered by the compared approaches (including AVal/-Modelan)

The criteria for comparison is the same one as the one defined in section 2.6.4. Table 7.1 compares the platform supported by the tools, the representation of both code and annotation elements in an application, and whether or not the tool provides full annotation support. Table 7.2 compares the tools on their support for constraint definition and validation. AVal/ModelAn is the only approach to support both code and annotation constraints, to provide both declarative and explicit constraint definition, and it is the only one to permit the definition of constraints both embedded in and external to the annotation framework. Finally, table 7.3 compares the support provided by the tools for the interpretation of annotated programs. Of the existing tools, only mTurnpike provides any kind of support for the compile-time interpretation of annotations. While the interpretation of annotated programs is not directly addressed by AVal/ModelAn, they are both integrated into the Spoon source-code processing framework. Also, the reified annotations provided by ModelAn ease the interpretation's implementation both at compilation and runtime.

As evidenced by tables 7.1, 7.2 and 7.3, AVal and ModelAn fill out the voids not covered by existing tools, thereby providing the annotation framework developer with the means to design, implement and validate and interpret annotation frameworks.

7.3 Implemented Tools

For the realization of the approaches presented in this thesis, two tools were implemented: AVal and ModelAn

7.3.1 AVal

AVal implements the seven generic constraints identified in section 7.1.2 by defining an annotation framework that provides one annotation per constraint. In order to ease the development of further annotation constraints, AVal exposes an API with which the framework may be extended. This extension feature is used to incorporate in AVal the checking of OCL-defined constraints.

AVal is implemented in Java, using the Spoon processing framework. It was developed in 1.5 KSLoC and it is available from the Spoon gforge website[20] where it has been downloaded more than a thousand times.

7.3.2 ModelAn

ModelAn is an annotation framework that allows the definiton and generation of an annotaiton model from a set of annotation types. ModelAn defines three annotations, and it relies on SpoonEMF and the Eclipse Modeling Framework to generate the annotation models and their corresponding instances. An Eclipse plug-in that visualizes annotation models and allows their navigation was also developed.

[20] http://gforge.inria.fr/projects/spoon/

ModelAn is also implemented in Java, using the Spoon processing framework, SpoonEMF, and the EMF code generation facilities. It was developed in four modules, totaling 1.2KSLoC and one 1KSLoC for the visualization plug-in.

7.4 Perspectives

Our work opens up several perspectives for future work. In this section we discuss some of them. Perspectives are separated in two areas: the ones pertaining generic constraints, and the ones related to annotation models.

7.4.1 Generic constraints

Augment the number of Generic Constraints We have contributed seven generic constraints, both annotation and code wise. It is clear that they do not cover all possible constraints that arise in annotation framework development. Further analysis of existing annotation frameworks would certainly uncover other generic constraints, for example, constraints defining property methods as found in JPA (see section 6.3.5).

Extend Generic Constraints to Non-annotation Frameworks In addition to this, the idea of using an annotation framework to express constraints in frameworks is applicable to frameworks other than annotation-based ones. Indeed this is a domain actively researched on [EKKM08, CGQ+06], and the lessons learned on the development of AVal would be of use in this direction. In particular, the importance of relations between entities represented in the framework, since a large number of constraints are attached to these relations.

Constraint Dependencies One of the strengths of generic constraints, as we have defined them, is that they are orthogonal. That is, each annotation can be checked by itself, independently of other annotations present. Nevertheless, in checking related annotations, sometimes it is useful to share information between different constraint implementations. For example, in SaxSpoon, the `XMLParser` annotation requires its parameter to be an URL that points to a DTD file. In addition to this, the `HandlesStartTag` annotation placed on methods that handle the start of a particular tag, require the names of the parameters of said method to be consistent with the attributes defined by the tag. This attribute information is present in the DTD. Now, if one were to write AVal constraints annotations to check these properties, it would be necessary to parse the DTD twice, once for the `XMLParser` and again to check the `HandlesStartTag`. In addition to this, it would make no sense to check the `HandlesStartTag` if the DTD is already proved to be invalid. Therefore, further research into how to define and implement dependencies between AVal checkers is required.

7.4.2 Annotation Models

Enhance Annotation-to-Model Mapping So far, the extraction of the annotation model from the annotation types is straightforward. Annotation types are converted into EClassifiers, and `Association` annotations express relations. This interpretation, while it results in valid models, might be too close to the original definition of the annotation types. This means that the annotation model would inherit the workarounds to the language's restrictions for annotation definition. A higher-level annotation-to-model mapping would be interesting by, for example, ignoring annotations that only contain other annotations, mapping marker annotations to boolean attributes instead of relations, and using the `RequiresAny` AVal constraint as a heuristic to discover inheritance relations.

Annotations that only contain other annotations are used to get around Java's restriction of only allowing one instance of an annotation on a given code element. When translated to the annotation model, these "collection annotations" would be translated to an association with unbounded cardinality to the collected annotations. Marker annotations (those that define no annotation elements) are normally used to mark the presence of a characteristic (for example the `Oneway` annotation on `WebMethod`s). Their mapping to the annotation model would be better represented as a boolean attribute that marks its presence on a code element. Finally, when a set of annotation types have the `RequiesAny` to the same annotations, this can be a clue to a hidden super-annotation from which the required annotation inherit. This is the case of the `Entity` and `MappedSuperclass` in the JPA.

Complement Annotation Models So far, annotation models are only linked to a model of the code on which they lay. In general, the development of a complex application deals with artifacts other than source code. The existing constraints on annotations sometimes refer to these artifacts, as is the case with SaxSpoon and the DTD of the parsed XML files. Such constraints are not expressible with the current implementation of ModelAn, since they are neither in the annotation or code models. If one were to define a model of DTD's, and then to write a tool that would take a given DTD and generate a DTD's model instance (a *model contributor*), then it would be possible to check constraints like the one that states that the names of the parameters of methods annotated with `HandlesStartTag` must be consistent with the names of the attributes defined in the DTD. Such model contributors would greatly enhance the expressiveness of ModelAn's constraints, as well as help in giving the application programmer a complete view of his application.

Publications

International Journals

[1] Carlos Noguera and Renaud Pawlak. AVal: an extensible attribute-oriented programming validator for java. *Journal of Software Maintenance and Evolution: Research and Practice*, Volume 19 Issue 4:253 – 275, July 2007.

International Conferences

[2] Carlos Noguera and Laurence Duchien. Annotation framework validation using domain models. In *Fourth European Conference on Model Driven Architecture Foundations and Applications*, pages 48–62, Berlin, Germany, June 2008.

[3] Carlos Noguera, Ellen Van Paesschen, Carlos Parra, and Johan Fabry. Context distribution for supporting composition of applications in ubiquitous computing. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, pages 1647–1648, New York, NY, USA, 2008. ACM.

[4] Johan Fabry and Carlos Noguera. Abstracting connection volatility through tagged futures. In *Proceedings of the 2nd Ambient Intelligence Developments (AmI.d) Conference*, pages 2–12, 2007.

International Workshops

[5] Johan Fabry and Carlos Noguera. Is AOP equal to untangling and unscattering? questions from an ambient intelligence application scenario. In *I Latin American Workshop on Aspect-Oriented Software Development (LA-WASP'2007)*, Joao Pessoa, Paraiba - Brasil, October 2007.

[6] Coen De Roover, Theo D'Hondt, Johan Brichau, Carlos Noguera, and Laurence Duchien. Behavioral similarity matching using concrete source code templates in logic queries. In *PEPM '07: Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 92–101, New York, NY, USA, 2007. ACM.

[7] Carlos Noguera and Renaud Pawlak. Aval: an extensible attribute-oriented programming validator for java. In *SCAM '06: Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 175–183, Washington, DC, USA, 2006. IEEE Computer Society.

[8] Carlos Noguera and Renaud Pawlak. Open static pointcuts through source code templates. In *AOSD 2006 Workshop on Open and Dynamic Aspect Languages*, 2006.

[9] Renaud Pawlak, Carlos Noguera, and Nicolas Petitprez. Spoon: Program analysis and transformation in java. Technical Report 5901, INRIA, May 2006.

Bibliography

[Bar06] Olivier Barais. SpoonEMF, une brique logicielle pour l'utilisation de l'IDM dans le cadre de la réingénierie de programmes Java5. In *Journées sur l'Ingénierie Dirigée par les Modèles (IDM)*, June 2006. Poster. 16, 80

[BCC05] Klaas van den Berg, Jose Maria Conejero, and Suzanna Chitchyan. AOSD ontology 1.0. Technical report, AOSD-Europe Network of Excellence, May 2005. 18

[BCL+06] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The FRACTAL component model and its support in java: Experiences with Auto-adaptive and Reconfigurable systems. *Software: Practice and Experience*, 36(11-12):1257–1284, 2006. 70, 90

[Ben86] Jon Bentley. Programming pearls: little languages. *Communications ACM*, 29(8):711–721, August 1986. 12

[BK06] Christian Bauer and Gavin King. *Java Persistence with Hibernate*. Manning Publications, 2006. ISBN 978-1932394887. 2, 28, 147

[BKVV06] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/xt 0.16. components for transformation systems. In *ACM SIGPLAN 2006 Workshop on Partial Evaluation and Program Manipulation (PEPM '06)*, pages 95–99, Charleston, South Carolina, January 2006. ACM SIGPLAN. 14

[CGQ+06] Dolors Costal, Cristina Gómez, Anna Queralt, Ruth Raventós, and Ernest Teniente. Facilitating the definition of general constraints in uml. In *Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006*, pages 260–274, 2006. 17, 130, 155

[Chi95] Shigeru Chiba. A metaobject protocol for c++. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 285–299, October 1995. 15

[CK05] V. Cepa and S. Kloppenburg. Representing Explicit Attributes in UML. In *7th International Workshop on Aspect-Oriented Modeling (AOM)*, 2005. 18, 21, 25

[CM04] Vasian Cepa and Mira Mezini. Declaring and enforcing dependencies between.NET custom attributes. In Gabor Karsai and Eelco Visser, editors, *Generative Programming and Component Engineering: Third International Conference, GPCE 2004, Vancouver, Canada, October 24-28, 2004. Proceedings*, volume 3286 of *Lecture Notes in Computer Science*, pages 283–297. Springer, 2004. 23, 63

[Cor06a] James R. Cordy. Source transformation, analysis and generation in txl. In *PEPM '06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 1–11, New York, NY, USA, 2006. ACM Press. 14

[Cor06b] James R. Cordy. The txl source transformation language. *Science of Computer Programming*, 61(3):190–210, 2006. 14

[Der05] Dirk Derrider. *A Concept-Centric Environment for Software Evolution in an Agile Context*. PhD thesis, Vrije Universiteit Brussel, 2005. 32

[DFFV06] Zoé Drey, Cyril Faucher, Franck Fleurey, and Didier Vojtisek. *Kermeta language reference manual*, 2006. 16

[EGHT94] David Evans, John Guttag, James Horning, and Yang Meng 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. 19

[EH07] Torbjörn Ekman and Görel Hedin. The jastadd extensible java compiler. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 1–18, New York, NY, USA, 2007. ACM. 14

[EKKM08] Michael Eichberg, Sven Kloppenburg, Karl Klose, and Mira Mezini. Defining and continuous checking of structural program dependencies. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, pages 391–400, New York, NY, USA, 2008. ACM. 19, 130, 155

[EL02] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, January/February 2002. 19

[EMOS04] M. Eichberg, M. Mezini, K. Ostermann, and T. Schafer. Xirc: a kernel for cross-artifact information engineering in software development environments. *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, pages 182–191, Nov. 2004. 22

[ESM05] Michael Eichberg, Thorsten Schäfer, and Mira Mezini. Using Annotations to Check Structural Properties of Classes. In Maura 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. 22

[FEBF06] Jean-Marie Favre, Jacky Estublier, and Mirelle Blay-Fornarino, editors. *L'ingénierie dirigée par les modèles.* Informatique et Systèemes d'Information. Lavoisier – Hermes Science, 2006. 17

[FJ97] Matteo Frigo and Steven G. Johnson. FFTW: The fastest fourier transform in the west. Technical Report MIT-LCS-TR-728, MIT LCS, 1997. 20

[FN07a] Johan Fabry and Carlos Noguera. Abstracting connection volatility through tagged futures. In *Proceedings of the 2nd Ambient Intelligence Developments (AmI.d) Conference*, pages 2–12, September 2007. 16

[FN07b] Johan Fabry and Carlos Noguera. Is AOP equal to untangling and unscattering? questions from an ambient intelligence application scenario. In *I Latin American Workshop on Aspect-Oriented Software Development (LA-WASP'2007)*, Joao Pessoa, Paraiba - Brasil, October 2007. 19

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, third edition, May 2005. 2, 29, 51, 55, 147

[GL04] Samuel Z. Guyer and Calvin Lin. Broadway: A compiler for exploring the domain-specific semantics of software libraries. In *Proceedings of the IEEE*, 2004. 20

[Hed97] Görel Hedin. Attribute extensions - a technique for enforcing programming conventions. *Nord. J. Comput*, 4(1):93–122, 1997. 19

[HH04] Erik Hilsdale and Jim 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. 19

[Joh78] Stephen Johnson. Lint, a C Program Checker, programmer's manual, AT&T bell laboratories, 1978. 19

[Joh97] Ralph E. Johnson. Frameworks = (components + patterns). *Communications ACM*, 40(10):39–42, 1997. 1, 2, 147, 148

[Jéz08] Jean-Marc Jézéquel. Model driven design and aspect weaving. *Software and Systems Modeling*, 2008. 17

[KBV08] Lennart C. L. Kats, Martin Bravenboer, and Eelco Visser. Mixing source and bytecode. A case for compilation by normalization. In G. Kiczales, editor, *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA 2008)*, volume 43–10 of *Lecture Notes in Computer Science*, pages 91–108, Nashville, Tenessee, USA, October 2008. ACM Press. 14

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In Jørgen Lindskov Knudsen, editor, *ECOOP*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer, 2001. 2

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997. 18

[KM05] Gregor Kiczales and Mira Mezini. Separation of concerns with procedures, annotations, advice and pointcuts. In Andrew P. Black, editor, *ECOOP*, volume 3586 of *Lecture Notes in Computer Science*, pages 195–213. Springer, 2005. 19

[LBCO04] Christian Lengauer, Don Batory, Charles Consel, and Martin Odersky, editors. *Domain-Specific Program Generation*, chapter From a Program Family to a Domain-Specific Language, pages 19–28. Springer-Verlag, 2004. 13

[LH01] Andreas Ludwig and Dirk Heuzeroth. Metaprogramming in the large. In *GCSE '00: Proceedings of the Second International Symposium on Generative and Component-Based Software Engineering-Revised Papers*, pages 178–187, London, UK, 2001. Springer-Verlag. 15

[LX07] Jessy Liberty and Donald Xie. *Programming C#*. O'Reilly, third edition, 2007. 23

[MB02] W Scott Means and Michael A Bodie. *The Book of SAX: The Simple API for XML*. No StarchPress, 2002. 33

[MFK04] Matthew Flatt Matthias Felleisen, Robert Bruce Findler and Shriram Krishnamurthi. Building little languages with macros. *Dr. Dobb's Journal*, April 2004. 1, 147

[MHS05] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, 2005. 13, 14

[MJCH08] Martin Monperrus, Jean-Marc Jézéquel, Joël Champeau, and Brigitte Hoeltzener. A model-driven measurement approach. In *In Proceedings of ACM/IEEE 11th International Conference on Model Driven Engineering Languages and Systems (MoDELS 08)*, Toulouse, France, October 2008. 16

[MK06] Linda De Michel and Michael Keith. *Enterprise JavaBeans, Version 3.0*. Sun Microsystems, May 2006. JSR-220. 2, 22, 45, 105, 147, 148

[MM03] Jishnu Mukeji and Joaquin Miller. *MDA Guide*. OMG, 2003. Version 1.0.1. 17

[ND08] Carlos Noguera and Laurence Duchien. Annotation framework validation using domain models. In *Fourth European Conference on Model Driven Architecture Foundations and Applications*, pages 48–62, Berlin, Germany, June 2008. 4, 150

[NP06] Carlos Noguera and Renaud Pawlak. Aval: an extensible attribute-oriented programming validator for java. In *SCAM '06: Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 175–183, Washington, DC, USA, 2006. IEEE Computer Society. 4, 150

[NP07] Carlos Noguera and Renaud Pawlak. AVal: an extensible attribute-oriented programming validator for java. *Journal of Software Maintenance and Evolution: Research and Practice*, Volume 19 Issue 4:253 – 275, July 2007. 4, 55, 150

[Obj04] Object Management Group. *Unified Modelling Language Infrastructure*, November 2004. Version 2.1.2. 17, 18

[Obj06] Object Management Group. *Object Constraint Language Specification*, 2006. Version 2.0. 17

[Par76] David Lorge Parnas. On the design and development of program families. *Software Engineering, IEEE Transactions on*, SE-2(1):1–9, March 1976. 13

[Paw05] Renaud Pawlak. Spoon: annotation-driven program transformation — the AOP case. In *AOMD '05: Proceedings of the 1st workshop on Aspect oriented middleware development*, pages 1–6. ACM Press, 2005. 3, 15, 31, 149

[PNP06] Renaud Pawlak, Carlos Noguera, and Nicolas Petitprez. Spoon: Program analysis and transformation in java. Technical Report 5901, INRIA, May 2006. 3, 15, 80, 91, 149

[Pro02] Jeff Prosise. *Programming Microsoft .Net*. Microsoft Press, 2002. 23

[Ray03] Eric S. Raymond. *The Art of UNIX Programming*. Professional Computing Series. Addison Wesley, 2003. 12

[RM06] Awais Rashid and Ana Moreira. Domain models are not aspect free. In *International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 155–169. Springer, 2006. 17

[RPPM06] Romain Rouvoy, Nicolas Pessemier, Renaud Pawlak, and Philippe 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. 2, 70, 91, 147

[SIP+05] Friedrich Steimann, Fachbereich Informatik, Lehrgebiet Programmiersysteme, Fernuniversität In Hagen, and D-Hagen. Domain models are aspect free. In *Proc. MODELS 2005*, pages 171–185. Springer, 2005. 17

[SPDC06] Lionel Seinturier, Nicolas Pessemier, Laurence Duchien, and Thierry Coupaye. *Component-Based Software Engineering*, volume Volume 4063/2006, chapter A Component Model Engineered with Components and Aspects, pages 139–153. Springer Berlin, 2006. 16

[Str05] Bjarne Stroustrup. A rationale for semantically enhanced library languages. In *Library-Centric Software Design LCSD'05*, October 2005. 20

[SY02] Mati Shomrat and Amiram 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. 19

[TCKI00] M. Tatsubori, S. Chiba, M.-O. Killijian, and K. Itano. OpenJava: A Class-Based Macro System for Java. In *Reflection and Software Engineering, Papers from OORaSE 1999, 1st OOPSLA Workshop on Reflection and Software Engineering*, volume 1826 of *Lecture Notes in Computer Science*, pages 117–133. Springer Verlag, Denver, Colorado, USA, 2000. 15, 32

[vdBKV01] M. van den Brand, P. Klint, and J. Vinju. Term rewriting with traversal functions. Technical report, Centrum Wiskunde & Informatica (CWI), 2001. 14

[vDHK96] Arie van Deursen, Jan Heering, and Paul Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific Publishing Co., 1996. 14

[vDKV00] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, June 2000. 12

[Vel98] Todd 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. 20

[VG98] Todd L. Veldhuizen and Dennis 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. 20

[WRO04] Craig Walls, Norman Richards, and Rickard Oberg. *XDoclet in Action*. Manning Publications, 2004. 91

[WS05] Hiroshi Wada and Junichi Suzuki. Modeling turnpike frontend system: A model-driven development framework leveraging UML metamodeling and attribute-oriented programming. In *Model Driven Engineering Languages and Systems, 8th International Conference, MoDELS 2005, Montego Bay, Jamaica, October 2-7, 2005, Proceedings*, pages 584–600, 2005. 21

[Zot05] Brian Zotter. *Web Services Metadata for the Java Platform, Version 2.0.* BEA Systems, June 2005. JSR-181. 2, 97, 147, 148

Appendixes

A

Formalization of Generic Constraints

A.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:

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

Having defined the tools to express ASTs and annotations we can define the notion of scope of an AST node:

$scope_N : N \rightarrow Set(N)$

$scope_N(n : N) := \{x : N | n <_N x\}$

And the scope of an annotation instance:

$scope_A : A \rightarrow Set(A)$

$scope_A(a : A) := \{b : A | target(b) \in scope_N(target(a)\}$

A.2 Annotation-wise Validations

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

`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`. 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 a constraint 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 Annotation Framework. 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 Annotation Framework presented Section 6.1 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)`.

`Unique`

If a and b are instances of the same annotation, and an element i of this definition has the *Unique* annotation, then, two AST nodes carrying a and b have the same value for the element i, they must be the same node.

$$\frac{type(a), type(b) <: Annotation \wedge def(a) = def(b) \qquad u : Unique \in annot(def(a.i))}{\forall n_1, n_2 \in AST\ (a \in annot(n_1) \wedge b \in annot(n_2) \wedge a.i = b.i \rightarrow a = b \wedge n_1 = n_2)}$$

A.3 Code-wise Validations

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

`Target`

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.

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

`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[21] 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.

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

[21] for methods returning `void`, a *Void* type is used

B

Résumé en français

L'art de la programmation consiste à prendre des concepts dans un domaine donné et à les projeter dans des concepts offerts par un langage de programmation. Par exemple, un *client* est représenté par une *classe* dans un langage à objets. La distance sémantique entre les concepts existants dans le domaine du problème et ceux offerts par le langage de programmation constitue un fossé sémantique. Les façons d'établir le lien entre le domaine et sa représentation dans un langage de programmation sont toujours des sujets de recherche actuels. Lorsque l'on développe un langage de programmation, on se doit d'équilibrer la généralité des concepts offerts par le langage en fonction de la largeur voulue du fossé sémantique. Les langages de programmation qui offrent des concepts trop généralistes sont difficiles à utiliser, alors que les langages de programmation qui proposent des concepts très spécifiques ont une applicabilité limitée. Une manière de trouver un équilibre est de développer un langage généraliste qui fournit des extensions spécifiques à un domaine, de sorte que les concepts du programme restent génériques, mais que les concepts plus spécifiques peuvent être exprimés, tel est le cas, par exemple, avec les macros de *Common Lisp* [MFK04]. Dans le langage Java, de telles extensions sont possibles par l'inclusion d'*annotations*. Les annotations sont des méta-données associées aux éléments de programme. Elles apportent une sémantique spécifique à un domaine. En quelque sorte, elles sont similaires aux directives de compilation ou *Pragmas*, sauf que la sémantique de ces annotations n'est pas définie par le langage, mais par un moteur externe d'interprétation. Le fait que des annotations soient accompagnées d'un moteur d'interprétation les rend plus proches des cadres logiciels que des directives de pré-processeur. Un cadre logiciel est défini par Johnson [Joh97] comme *a reusable design of all of parts of a system that is represented as a set of abstract classes and the way their instances interact*. Dans le cas des cadres à base d'annotations, à la place d'étendre des classes abstraites, le concepteur d'applications annote son programme et la manière avec laquelle les annotations seront interprétées est similaire aux traitements qui résultent de l'interaction avec les instances du cadre.

L'association de méta-données à des éléments de programme est actuellement utilisée par plusieurs cadres logiciels de grande ou petite taille [RPPM06, MK06, Zot05, BK06]. Néanmoins, leur développement est encore, en grande partie, effectué de façon ad-hoc . Les annotations, telles que définies dans le langage Java [GJSB05], permettent au développeur d'annoter certains éléments d'un programme (classes, paquets, méthodes, etc.) pour dé-

clarer qu'elles respectent une sémantique spécifique à un domaine. L'utilisation des annotations fournit un certain nombre d'avantages. D'abord, elles fournissent un mécanisme d'interaction aditionnelle entre l'application et le cadre logiciel qui propose les annotations. En second lieu, elles améliorent le découplage entre l'interface du cadre (représentée sous la forme d'annotations) et son exécution (l'interprétation des annotations). Ce découplage rend les annotations attractives pour le développement des caractéristiques qui doivent être réalisées par des tiers. Par exemple, l'API de persistance de Java (une partie de la spécification Java EJB3 [MK06]), ou la spécification de méta-données de services Web pour Java [Zot05]. Troisièmement, en faisant en sorte que les éléments annotés déclarent qu'ils offrent une sémantique spécifique à un domaine en plus de celle fournie par le langage, les annotations peuvent alors être utilisées pour étendre le langage Java (comme cela est fait avec AspectJ5).

B.1 Développement de cadres d'annotations

Nous définissons un cadre d'annotations comme un cadre logiciel [Joh97] qui propose des annotations en tant que moyens d'interaction. Une cadre d'annotation est composé de deux parties: l'ensemble des types d'annotation et le moteur d'interprétation. Les types d'annotation correspondent à l'interface du cadre d'annotations. Ils représentent les concepts que l'utilisateur du cadre doit étendre avec ses propres annotations afin d'en utiliser les services. Les annotations en Java sont des types similaires aux interfaces, qui contiennent un certain nombre d'éléments d'annotation. Ils fonctionnent comme des champs statiques dans les classes. L'utilisateur d'annotations est responsable de mettre en correspondance les concepts de son application tels que les classes, paquets, méthodes ou champs avec les concepts fournis par le cadre d'annotations. Cette mise en correspondance est faite explicitement en décorant le code source de l'élément ciblé avec une annotation. En faisant ceci, l'utilisateur d'annotations doit être conscient de la sémantique des types d'annotation qu'il emploie, et à son tour, le cadre d'annotations doit fournir les erreurs et les avertissements opportuns toutes les fois que le développeur viole les contraintes des annotations.

En plus du défi de vérifier (et spécifier) la sémantique des types d'annotation, le développeur du cadre d'annotations doit surmonter d'autres défis en concevant et en réalisant le cadre. Les types d'annotation, comme déjà dit, représentent un certain nombre de concepts qui proviennent du domaine que le cadre représente. De ce modèle de domaine vont provenir certaines des contraintes sur les types d'annotation. En concevant les types d'annotation pour ce cadre, le développeur est confronté aux limitations imposées par le langage de programmation dans la définition des types d'annotation. Il s'agit, par exemple, du fait que les annotations peuvent seulement définir un nombre limité de types pour leurs éléments, ou encore que l'héritage entre des annotations ainsi que les associations entre annotations ne soient pas permis. Ceci constitue le défi numéro `I` dans le développement des cadres d'annotations. Le défi numéro `II` est celui de la définition et de la vérification des contraintes d'annotation. Deux types de contraintes existent pour des annotations: ceux qui définissent les relations entre les types d'annotation, et ceux qui définissent les relations entre les annotations et les éléments du code source sur lesquels

elles sont placées. Le premier type de contraintes est enraciné dans le modèle de domaine que les annotations représentent ; tandis que le second vient de la mise en correspondance du modèle de domaine avec le modèle du code (c'est-à-dire de l'arbre de syntaxe abstraite ou AST) du langage Java. La définition de ces deux types de contraintes représente le défi numéro `III`

Finalement, le défi numéro `IV` traite de l'interprétation des programmes annotés. Les annotations en Java ne définissent pas une sémantique. Leur comportement est donné par leur interprétation par le cadre d'annotations. Les programmes annotés peuvent alors, être interprétés au moment de la compilation, en transformant le code source du programme annoté en code source sans annotations, ou encore au moment de l'exécution en utilisant l'API de réflexion de Java, le traitement du programme est alors changé car dirigé par la présence des annotations. Dans l'un comme dans l'autre cas, pour interpréter les annotations, il est souhaitable de les réifier de sorte à ce qu'elles ressemblent à leurs modèles du domaine d'origine. Actuellement, aucun outil de traitement des annotations n'offre de telles réifications, laissant la tâche au développeur du cadre d'annotations.

Pour récapituler, les quatre défis identifiés dans le développement des cadres d'annotations sont :

I la représentation des concepts du domaine comme des types d'annotation,

II la mise en correspondance des types d'annotation avec les éléments du code,

III la définition des contraintes pour la validation des programmes annotés,

IV le réification des annotations pour leur interprétation.

B.2 Proposition

Nous proposons de résoudre ces défis dans le développement de cadres d'annotations en deux étapes: la définition de contraints génériques et la définition de modèles d'annotation. D'abord, nous analysons les contraintes dans les cadres existants, puis nous extrayons un certain nombre de contraintes génériques. Ces contraintes peuvent alors être paramétrées par des développeurs de cadres d'annotations pour spécifier leurs types d'annotation. En second lieu, nous empruntons des concepts l'ingénierie dirigée par des modèles pour créer les modèles d'annotation qui sont d'un niveau d'abstraction plus élevé que les types simples d'annotation. Les modèles d'annotation permettent la définition de la mise en correspondance des types d'annotation avec les éléments de code et fournissent une base pour la réification des annotations dans les applications. L'exécution des deux approches est basée sur le moteur de transformation Spoon[PNP06, Paw05]. Spoon est approprié en particulier pour l'analyse et la transformation des annotations, puisqu'il est fondé sur la syntaxe de Java5, et qu'il fournit des processeurs[22] spéciaux pour des éléments de code annotés.

[22] visiteurs de l'AST du programme

B.2.1 Contraintes génériques

Nous classifions les contraintes dans les cadres d'annotations en deux grands groupes: les contraintes liées aux annotations, qui traitent des contraintes sur l'ensemble des valeurs possibles pour les éléments des annotations et sur les relations entre les annotations, et les contraintes liées au code, qui traitent des propriétés que les éléments du code demandent afin de pouvoir être annotées avec un type particulier d'annotation. Pour chaque groupe d'annotations, nous définissons un certain nombre de contraintes, et nous définissons comment elles se composent afin de spécifier un type d'annotation.

Afin de pouvoir employer ces contraintes génériques, nous avons mis en place un cadre d'annotations pour la définition et la validation des contraintes d'annotation du domaine appelé AVal[23] [NP07, NP06]. Chacune de ces contraintes génériques est représentée par un type d'annotation. Le développeur du cadre d'annotations utilise AVal en positionnant une annotation contrainte sur les types d'annotation de son cadre. AVal est fondé sur le cadre logiciel Spoon pour vérifier les contraintes sur les applications. Cette validation est faite en utilisant une représentation de l'AST du programme fourni par Spoon.

Les annotations d'AVal fournissent un moyen générique, réutilisable et déclaratif pour spécifier les contraintes d'un cadre d'annotations. En plus, AVal fournit également un moyen de vérifier que les programmes annotés sont conformes aux spécifications des cadres d'annotations qu'ils utilisent.

B.2.2 Modèles d'annotation

Au moment de la conception, le développeur du cadre d'annotations doit faire face aux restrictions que le langage Java impose. La première de ces restrictions est celle qui empêche de définir des relations entre types d'annotation. Les relations entre les annotations sont définies à partir du modèle de domaine qu'elles représentent. De telles relations sont souvent nécessaires pour définir des contraintes et pour interpréter les applications annotées. Pour cette raison, nous proposons d'ajouter aux types d'annotation avec la notion d'*association*. Comme pour les contraintes génériques, nous mettons en place cette extension par un cadre d'annotations appelé ModelAn[24] [ND08]. ModelAn définit une méta-annotation (`Association`) qui, une fois placée sur un type d'annotation, représente une association avec un autre type d'annotation. Le graphe résultant des types d'annotation et de ces associations est ce que nous appelons un *modèle d'annotations*.

Comme les modèles d'annotation contiennent des informations qui sont plus proches de celles existantes dans le modèle de domaine, le défi numéro `I` est résolu par les modèles d'annotations en rendant plus faible les relations entre les annotations, c'est-à-dire la restriction la plus ennuyeuse quand on passe du modèle du domaine à un ensemble de types d'annotation. Des entités représentant des types d'annotation dans un modèle d'annotations sont associées aux éléments de l'AST du langage Java de la même manière que des annotations sont liées aux éléments du programme sur lesquels elles sont placées. Cette association peut être qualifiée par une requête qui représentera l'élément de code sur lequel des annotations seront censées être placées. Ceci relève le défi numéro `III`,

[23] en anglais pour *Annotation Validation*
[24] en anglais pour *Modeling Annotations*

puisqu'en employant ces requêtes, le développeur du cadre d'annotations peut exprimer la mise en correspondance entre les types d'annotation et les éléments du code des applications qui les utiliseront.

En conclusion, les entités dans les types d'annotation sont employées pour générer leur réification (défi numéro `IV`). Avec cette réification des types d'annotation, le développeur de cadre peut alors interpréter des applications annotées sans avoir recours à la représentation de l'annotation sur l'AST. En plus, comme ModelAn et les annotations réifiées sont basés sur Spoon, ces dernières sont accessibles depuis les processeurs d'annotation de Spoon.

Ainsi, ModelAn et AVal fournissent des moyens pour venir à bout des quatre défis identifiés dans le développement de cadres d'annotation. Comme ModelAn et AVal traitent des annotations au niveau du code source des programmes, les développeurs peuvent spécifier et modéliser des annotations indépendamment de leur politique de conservation (si les annotations sont traitées uniquement dans le code source, dans le byte-code ou au moment de l'exécution). En outre, les annotations réifiées peuvent être utiles que ce soit dans le cas d'une interprétation au moment de la compilation (utilisant Spoon) ou au moment de l'exécution; bien que cette dernière propriété ne soit pas encore mise en application.

Ce mémoire de thèse est composé de sept chapitres. Le chapitre 1 sert d'introduction, le chapitre 2 décrit les autres travaux dans des domaines proches comme l'ingénierie dirigée les modèles, les langages dédies et la programmation orientée aspect. Le chapitre 3 propose une introduction aux cadres logiciels à base d'annotations. Le chapitre 4 définis des contraintes génériques pour la spécification et validation des cadres d'annotations, tandis que le chapitre 5 propose des modèles d'annotation pour le développement des cadres d'annotation. Les propositions des chapitres 4 et 5 sont mise en œuvre dans les cadres d'annotation existantes en le chapitre 6 pour valider son utilité. Enfin, le chapitre 7 présente un bilan des travaux de cette thèse et donne des perspectives de recherche.

Les prochaines sections présent les contributions et perspectives de cette thèse.

B.3 Contributions

L'objectif de cette thèse est donc de fournir des outils et des techniques pour l'aide au développement de cadres d'annotation. Afin de faire ceci, nous avons analysé la conception et le développement des cadres d'annotation et avons identifié quatre défis auxquels le développeur de cadre d'annotations doit répondre :

I la représentation des concepts de domaine comme types d'annotation,

II la mise en correspondance des types d'annotation avec les éléments de code,

III la définition de contraintes pour la validation de programmes annotés,

IV le réification des annotations pour leur interprétation.

Les contributions sur les deux parties de l'approche, contraintes génériques et modèles d'annotation, sont détaillés ci-dessous.

B.3.1 Contraintes génériques

Nous avons défini une classification des contraintes présentes dans des cadres d'annotation en fonction du sujet de la contrainte. Deux classes de contraintes sont définies : celles liées aux annotations pour les contraintes qui traitent des propriétés des annotations et de leurs éléments ; et celles liées au code, pour les contraintes qui traitent des propriétés du code sur lequel des annotations sont placées. En analysant les cadres d'annotations existants, nous avons identifié sept contraintes génériques et nous avons fondé sur celle-ci la mise en œuvre du cadre d'annotation AVal. AVal possède les propriétés suivantes:

Générique Les annotations d'AVal peuvent être utilisées sur des cadres d'annotations industrielles, et des contraintes communes peuvent être exprimées avec une combinaison d'annotations AVal. En plus, des contraintes génériques peuvent être employées pour extraire le modèle d'annotation du cadre d'annotations. Par exemple, si une annotation est requise par une autre, ceci peut être vu comme une association optionnelle entre elles.

Cependant, puisque les contraintes génériques doivent rester génériques, elles ne peuvent pas adresser des contraintes spécifiques à un cadre donné d'annotations. Pour cela, l'extensibilité du cadre AVal permet facilement la prise en compte de nouvelles contraintes.

Déclaratif Les annotations contraintes d'AVal sont des spécifications déclaratives pour les types d'annotation définis dans un cadre. Le développeur du cadre n'a pas besoin de préciser explicitement la façon dont les contraintes seront vérifiées dans les programmes annotés.

Paramétrisable La plupart des annotations contraintes dans AVal peuvent être paramétrées afin d'adapter les messages présentés à l'utilisateur lorsque les contraintes sont violées. Il est possible, aussi, d'attacher une transformation de code à chaque annotation AVal qui fournira un moyen de corriger l'erreur. La possibilité d'adapter les messages d'erreur est essentielle dans AVal, étant donné que ses contraintes sont génériques, les messages d'erreur par défaut sont également génériques et donnent peu de indices à l'utilisateur sur la cause de l'erreur ou sur les étapes qui doivent être suivies pour leur résolution. La connaissance des deux configurations est à la charge du développeur de cadre d'annotations.

Extensible AVal fournit un API qui permet aux développeurs de cadre d'annotations de construire leurs propres annotations contraintes lorsque cela est nécessaire.

Composable Nous décrivons la manière dont des annotations contraintes se composent tout en respectant les contraintes du langage Java sur le placement des annotations sur un élément de code.

Augmentation de la compréhension du code L'inclusion de contraintes AVal sur des types d'annotation augmente leur compréhension, puisque les règles d'utilisation des

annotations sont explicitement énoncées. Ceci facilite leur utilisation par les développeurs d'applications

L'approche de contraintes génériques d'AVal, comparée à d'autres approches dans la littérature, est la seule à tenir compte des contraintes liées aux annotations et au code. En plus, elle est la seule à fournir un mécanisme d'extension pour définir de nouvelles contraintes, et une personnalisation permet la définition de messages d'erreur et de transformations correctrices.

B.3.2 Modèles d'annotation

Nous proposons une extension des types d'annotation Java par l'expression de relations entre eux. En utilisant cette extension, nous repondons aux trois autres défis, tout en complétant le défi III résolu par AVal. Le défi I, la représentation du domaine des concepts comme annotations est adressée en construisant un modèle d'annotation des types d'annotation et a leurs relations. La mise en relation entre les types d'annotation et les éléments de code, défi II, est prise en considération en ajoutant aux types d'annotation des requêtes OCL qui résolvent les éléments sur lesquels ils sont tracés. Nous prolongeons également AVal, de sorte que des contraintes puissent être définies comme des expressions OCL sur ce modèle d'annotation. Finalement, les éléments définis dans le modèle d'annotation sont transformés en classes Java qui servent d'annotations réifiées dans l'application. Ces annotations réifiées peuvent être utilisées pour interpréter le code source de l'application (défi IV).

Les modèles d'annotation contiennent les propriétés suivantes :

Définition complémentaire du type d'annotation L'addition des associations aux types d'annotation les rend plus proche du modèle de domaine qu'ils représentent. En définissant des assocations, le développeur de cadre peut faire plus facilement la transition entre le modèle de domaine et le développement des types d'annotation.

Abstraction de plus haut niveau pour le cadre d'annotation Le modèle de domaine permet une abstraction de plus haut niveau pour la définition du cadre d'annotations que celle fournie par le code source seul. Le modèle de domaine permet l'expression explicite des interactions entre les types d'annotation. Ceci fait du modèle d'annotation un outil utile pour la documentation du cadre.

Rapprochement entre annotations et modèles Les modèles d'annotation permettent également le rapprochement entre les annotations et les modèles. Le concepteur de cadre d'annotation peut alors tirer des avantages des outils et techniques existants dans l'ingénierie dirigée par les modèles et les utiliser dans le cadre du développement orienté annotation. Dans cette thèse, nous exploitons ce rapprochement en réalisant des contraintes et des requêtes d'annotation en OCL et également en renforçant l'outillage du cadre logiciel Eclipse par la réalisation de la réification des annotations.

Modèle extensible Bien qu'un modèle d'annotation représente les types d'annotation de cadres simples, plusieurs instances différentes de modèles d'annotation peuvent coexister dans une même application. Quand une application emploie différents cadres d'annotation, leurs instances de modèle d'annotation sont fusionnées dans un modèle unique qui inclut la représentation de toutes les annotations ainsi que le code de l'application.

Vue multidimensionnelle d'une application Quand plusieurs modèles d'annotation coexistent dans une application, chaque instance de modèle d'annotation représente une vue spécifique à un domaine. Si une application complète emploie un cadre d'annotations pour la persistance et un autre pour la navigation dans les pages Web, alors il y aura une instance de modèle d'annotation qui définit le graphe de navigation de l'application et un autre qui montre le schéma de persistance.

Vérification de contraintes Nous prolongeons les annotations contraintes génériques d'AVal par une annotation qui prend en paramétre une requête OCL qui sera évaluée par l'instance du modèle d'annotation. L'utilisation d'OCL pour exprimer des contraintes n'exige pas la connaissance l'API Spoon, et devrait donc être plus accessible aux développeurs de cadre d'annotations.

Définition des valeurs complexes par défaut En employant des requêtes OCL audessus du modèle d'annotation, nous permettons, au développeur de cadre, l'écriture de valeurs complexes par défaut pour les éléments d'annotation. Aucun autre instrument de développement d'annotation actuel ne permet ceci.

Relations explicites Annotation-Code Par l'utilisation de requêtes OCL, nous permettons également au développeur de cadre d'annotations de définir la relation entre les annotations et le code sur lequel elles sont placées. Ceci rend les annotations proches des aspects en leur fournissant un mécanisme de point de coupe. Ces relations sont également utilisées pour exprimer des annotations implicites présentes dans des cadres d'annotations complexes.

B.4 Perspectives

Notre travail ouvre plusieurs perspectives pour des travaux futurs. Dans cette section nous discutons certains d'entre eux. Ces perspectives sont séparées en deux grandes parties : celles concernant les contraintes génériques et celles se rapportant aux modèles d'annotation.

B.4.1 Contraintes génériques

L'augmentation du nombre de contraintes génériques Nous avons défini sept contraintes génériques qui s'appliquent soit côté annotation soit côté code. Il est clair

qu'elles ne couvrent pas toutes les contraintes possibles qui surgissent lors du développement de cadre d'annotations. L'analyse approfondie des cadres existants d'annotation permettrait certainement de découvrir d'autres contraintes génériques, comme par exemple, des contraintes définissant des méthodes de propriété comme nous avons trouvées dans JPA.

Extension des contraintes génériques aux cadres sans annotation En plus de ceci, l'idée de l'utilisation d'un cadre d'annotations pour exprimer des contraintes dans des cadres logicels est applicable aux cadres autres que ceux basés sur les annotations. En effet, la validation des cadres logicielles est un domaine de recherche actif [EKKM08, CGQ+06], et l'expérience acquise avec le développement d'AVal serait utile dans cette direction. Par exemple, avec AVal les relations entre les entités représentées dans le cadre sont importantes à cause du grand nombre des contraintes attachées à ces relations; et aucune des approches mentionnes dans la littérature ne fait référence à ces relations.

Dépendances entre contraintes Une des forces de la notion de contrainte générique, telle que nous l'avons définie, est que celles-ci sont orthogonales, c'est-à-dire que chaque annotation peut être vérifiée par elle-même, indépendamment des autres annotations présentes. Néanmoins, en testant des annotations dépendantes, il est parfois utile de partager l'information entre les différentes réalisations des contraintes. Par exemple, dans SaxSpoon, l'annotation de XMLParser exige que son paramétre soit une URL qui pointe dans une DTD. En plus de ceci, l'annotation `HandlesStartTag` placée sur les méthodes qui gèrent le début d'une étiquette *tag* particulière, exige que les noms des paramètres de ladite méthode soient compatibles avec les attributs définis par l'étiquette. Cette information d'attribut est présente dans la DTD. Maintenant, si on souhaitait d'écrire des annotations contraintes AVal pour vérifier ces propriétés, il serait nécessaire d'analyser la DTD deux fois, une fois pour le XMLParser et une fois pour le `HandlesStartTag`. De plus, il ne semblerait pas raisonnable de vérifier le `HandlesStartTag` si la DTD s'avère invalide. Par conséquent, il est nécessaire de mener d'autres investigations sur la façon de définir et d'instrumenter les dépendances entre les contrôleurs d'AVal.

B.4.2 Modèles d'annotation

Complément pour la mise en correspondance des annotations et du modèle L'extraction du modèle d'annotations à partir des types d'annotation a été clairement identifiée. Les types d'annotation sont convertis en *EClassifiers*, et les annotations `association` correspondent aux relations mises en place entre annotations. Cette interprétation, alors qu'elle a pour conséquence de la mise en place modèles valides, peut être vue comme trop proche de la définition originale des types d'annotation. Ceci signifie que le modèle d'annotation hériterait des contours des restrictions du langage de définition d'annotation. Une mise en correspondance de plus haut niveau entre annotations et modèle serait intéressante, par exemple, pour ignorer les annotations qui contiennent uniquement d'autres annotations, pour gérer la mise en correspondance des annotations marqueurs par des attributs booléens à la place des relations ou encore pour l'utilisation

des contraintes Aval `RequiresAny` comme heuristique de découverte des relations d'héritage.

Des annotations qui contiennent uniquement d'autres annotations sont utilisées pour détourner la restriction de Java qui permet une et une seule instance d'annotation sur un élément de code donné. Une fois traduites dans le modèle d'annotation, les *collections d'annotations* devraient être transformées en une association avec une cardinalité non bornée vers des annotations collectées. Les annotations marqueurs (ceux qui ne définissent aucun élément d'annotation) sont normalement employées pour marquer la présence d'une caractéristique (par exemple l'annotation `Oneway` sur `WebMethodS`). Leur mise en relation sur le modèle d'annotation serait mieux perçue par un attribut booléen qui marque la présence dans un élément du code. Finalement, quand un ensemble de types d'annotation applique `RequiesAny` sur les mêmes annotations, ceci peut être un indice d'une super annotation cachée dont l'annotation requise hérite. C'est par exemple le cas des annotations `Entity` et `MappedSuperclass` dans JPA.

Complément pour les modèles d'annotation Jusqu'ici, des modèles d'annotation étaient uniquement liés à un modèle de code sur lequel les annotations sont déployées. Généralement le développement d'une application complexe traite d'artefacts autres que le code source. Les contraintes existantes sur des annotations se rapportent parfois à ces artefacts, comme cela est le cas pour SaxSpoon et la DTD des fichiers XML parsés. De telles contraintes ne sont pas exprimables dans l'implémentation courante de ModelAn, puisqu'elles ne sont ni dans les modèles d'annotation ni dans le modèle de code. Si on définit un modèle de DTD et que l'on écrit alors un outil qui prendrait une DTD donnée et générait une instance du modèle de DTD (un *contributeur de modèle*), alors il serait possible de vérifier des contraintes comme celles qui déclarent que les noms des paramètres des méthodes annotées avec `HandlesStartTag` doivent être compatibles aux noms des attributs définis dans la DTD. De tels contributeurs de modèles augmenteraient considérablement l'expressivité des contraintes de ModelAn en donnant au programmeur d'application la possibilité d'avoir une vue complète de son application.