DOI: 10.15439/2015F173
ACSIS, Vol. 5

Source Code Annotations as Formal Languages

Milan Nosáľ, Matúš Sulír, and Ján Juhár
Department of Computers and Informatics
Technical University of Košice
Letná 9, 042 00 Košice, Slovakia
Email: milan.nosal@gmail.com, {matus.sulir,jan.juhar}@tuke.sk

***Abstract*—Attribute-oriented programming (source code annotations) is a program level marking technique that enables enrichment of program elements with custom metadata. In this paper we hypothesize that there is a correspondence between source code annotations and conventional formal languages in general. We analyze our observations about source code annotations from three aspects of language description: concrete syntax, abstract syntax, and semantics. The discussion provides evidence of the hypothesized correspondence and we use it as a basis for our definition of an *annotation-based language* (abbreviated: @L). However, the analysis also shows that compared to conventional formal languages, source code annotations have some specificities mainly connected to their binding to host program elements. The presented analysis contributes to the field of attribute-oriented programming by discussing the relationship between annotations and conventional formal languages, and by surveying relational idioms in annotations' usage that can be inspirational for annotations' authors.**

I. Introduction

ATTRIBUTE-ORIENTED PROGRAMMING (abbreviated: @OP) as a technique of marking source code elements with source code annotations [1], [2] became quite popular during the last decade, as is manifested by multiple frameworks, such as the Spring Framework. The annotations as a metadata format found the same popularity in the academic environment as well. As an example we can mention an annotation-based parser generator YAJCo, which uses annotations with an object-oriented language model for syntax [3] and references definition [4]. We can also recognize a research field dedicated to annotations. Most important examples include a book about attribute-enabled software development by Cepa [5] that summarizes his research in the field, multiple research articles about enforcing annotations' dependencies in source code such as work of Noguera et al. [1], or one of the first articles on the topic of how to use (or how not to use) annotations by Correia et al. [6].

In our previous work [7] we analyzed the correspondence between annotations and XML in the scope of configuration languages. The main manifestation of the correspondence was the discovery of a set of mapping patterns between annotations and XML. Using the discovered mapping patterns we showed that annotations and XML can be considered equivalent in terms of their expressibility. Based on our previous work, in this paper we want to examine the relationship of annotations and formal languages deeper. A formal language is defined by an alphabet and a set of formation rules (grammar[1]). We will try to show that the same can be applied to annotations.

Our work presented in [7] can be considered a case study on correspondence between annotations and a representative of a generic language, XML. It indicates that there is probably a correspondence between annotations and formal languages in general, as well. Therefore in this paper we hypothesize that *there is a correspondence between annotations and conventional formal languages*. We will provide an analysis of our observations supporting this hypothesis and discuss the annotations from three language description aspects, concrete syntax (abbreviated: CS[2]), abstract syntax (abbreviated: AS), and semantics. The evidence presented in this work indicates that we can use the formal language theory when we are working with annotations. In consequence, the application of the approaches from the language theory can provide benefits for annotations' authors and users. In related work in section IX we will discuss several other works that connected annotations with languages, however, none of them considered this correspondence a hypothesis and tried to prove it. We will conclude the paper in section X with a brief discussion of consequences of proving this hypothesis.

The contributions of this work are as follows:

- observations of corresponding characteristics between source code annotations and formal languages (sections III, VI, and VII),
- discussion of discrepancies between annotations and formal languages (and thus identification of the main specificity of annotations in comparison with formal languages, section IV),
- survey (overview) of relational idioms between annotations and between annotations and their host language that can help annotations' authors during designing the annotations (sections III-B, IV-B, and IV-D),
- discovery of reversed code-wise relations between annotations and their target program elements that emphasize the significance of annotations relation to their host language (section IV-C), and

This work was supported by VEGA Grant No. 1/0341/13 "Principles and Methods of Automated Abstraction of Computer Languages and Software Development Based on the Semantic Enrichment Caused by Communication".
This paper shares contents with the first author's dissertation thesis.

[1]Grammar [8] described by concrete syntax (including also the lexical syntax) and abstract syntax. Abstract syntax describes the structural restrictions between language concepts (words). Concrete syntax describes the actual representation of the language sentences in the given alphabet.

[2]Not to be confused with Counter-Strike.

- definition of *annotation-based language* (abbreviated: @L) and its aspects from the viewpoint of formal language theory (throughout the whole paper with a summary in section VIII).

II. Source Code Annotations

Before going into discussion about the correspondence we will clarify several essential terms connected with source code annotations. An *annotation-enabled language* (abbreviated: @EL) is any formal language that supports attribute-oriented programming. A language supports the attribute-oriented programming if its grammar (and therefore its parser too) allows adding custom declarative tags to annotate standard program elements. These tags have to be structured and therefore parsable by the parser (or by some additional tool, as in case of XDoclet[3] technology). An example of an @EL is Java programming language from version 1.5. In previous versions the attribute-oriented programming was supported by a 3rd party tool XDoclet. This means that if we add these tags to a valid host language sentence, it still *has* to be a completely valid host language sentence without any preprocessing or other manipulations. In other words, the @EL parser has to have an extension point, a grammar rule, that enables these tags. In case of Java annotations, they are a part of the Java language. In case of XDoclet, the XDoclet annotations are part of standard Java comments.

Annotations do not directly change the source code semantics. They only add metadata to source code. Annotations can be queried and processed on demand by frameworks or tools, or the program itself, thus indirectly changing program semantics.

Custom declarative tags supported by @EL are *source code annotations* (abbreviated: annotations). An annotation annotates an annotation-enabled program element[4]. E.g., a program element can be a method, a function, a class, a statement, etc., depending on language's programming paradigm. We will call the program element annotated by an annotation the *target host program element* (abbreviated: target element) of that particular annotation.

Source code annotations can be also dynamic as discussed by Noguera et al. [9] and Cazzola et al. [10]. Dynamic annotations can have properties that are evaluated dynamically when the annotations are processed (conventional static annotations are evaluated during compilation, just as constants). A dynamic annotation can have a property that is bound to some code property, e.g., a property that is bound to count of loop iterations cannot be evaluated during compile time. During runtime, when the loop is evaluated, the annotation will remember the iteration count and it can be queried. However, dynamic annotations cannot directly change program semantics as well.

[3]http://xdoclet.sourceforge.net/xdoclet/index.html

[4]Annotation-enabled program element is a program element that can be annotated. In some cases not all program elements can be annotated. E.g., in the Java language the `if` program element cannot be annotated by Java annotations (statements in general).

The relation between an annotation and its target element has to be expressed by in-place binding. In-place binding requires that an annotation is placed next to or directly into its target program element declaration. The placement has strict rules that are dependent on the host @EL grammar. In General-Purpose Languages (abbreviated: GPL) it is usually expressed by prepending an annotation directly before the program element declaration. Our definition of annotations is presented in Definition 1.

Definition 1. *Annotations are custom declarative structured tags in host @EL that are bound to host program elements using in-place binding. Annotations have to be parsable by standard @EL parser (or a 3rd party tool) that allows implementing semantics in form of a plug-in.*

What follows is an analysis of the correspondence between annotations and formal languages.

III. Abstract Syntax Correspondence

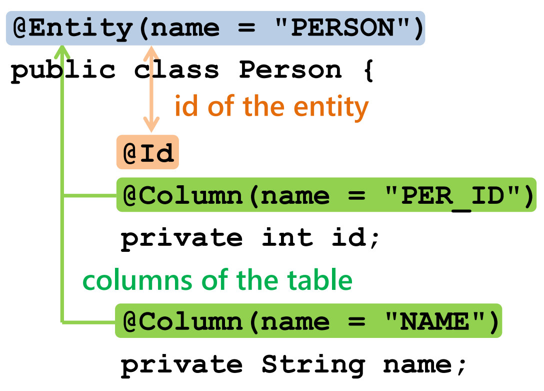
The main idea of the correspondence between a formal language and a set of related annotations stems from the fact that we can observe a structure (abstract syntax) in using annotations from the given set. The observed regularities are not accidental, and in all cases they are even enforced by the processing tools.

A. Structural Correspondence Example

Let us begin with an illustrative example based on Java Persistence API (abbreviated: JPA) annotations. JPA is an object-relational mapping specification for Java. In JPA annotations are used to specify mapping between Java classes and a relational database. In a simple example presented in Figure 1 an `@Entity` annotation is used to specify that the `Person` class is going to be a persisted entity, and `@Column` annotations specify the mapping of fields to table columns. JPA specification requires the user to use the `@Entity` annotation to include the class in the persistence management setup[5]. Without the `@Entity` annotation all the `@Column` and the `@Id` annotations would not be processed by any JPA compliant object-relational mapping (ORM) tool. This relationship is represented by green arrows in Figure 1. Another requirement is that each entity marked with the `@Entity` annotation has to be annotated by the `@Id` annotation to specify which of the fields represent a primary key. This relationship is highlighted in orange in Figure 1.

Considering the example from Figure 1 one can easily notice that the `@Entity` annotation and the `@Column` annotations mimic an abstract syntax tree (abbreviated: AST). Annotations and their properties are nodes, thus modelling a tree. A sketch of such an AST is shown in Figure 2. We also added a simplified AST of the host language to illustrate the binding of annotations to their target elements.

[5]One can alternatively use `@Embeddable`, or `@MappedSuperclass`, but those have slightly different semantics and are not important for the discussion.



```
@Entity(name = "PERSON")
public class Person {
    @Id
    @Column(name = "PER_ID")
    private int id;
    @Column(name = "NAME")
    private String name;
```

Fig. 1. Mapping of the Person entity class to database using JPA annotations

Now when we look at the given AST, we can easily devise a simple external domain-specific language[6] (abbreviated: DSL) that would to a degree copy the structure of these annotations. Snippet in Listing 1 can be an illustration of an external DSL that expresses the same information as annotations. Here we can see by an example that there is a correspondence between annotations and formal languages.

Listing 1. Person entity definition

```
Entity "PERSON" {
  Id Column  "PER_ID"
  Column     "NAME"
}
```

B. Idioms in Structural Relations

The example shown in section III-A is just an illustrative example. In practice, however, we can find multiple structural stereotypes (idioms) in annotations' usage that support our hypothesis. Each *idiom specifies common structural relationship between annotations* (annotation-based language concepts) and as such defines a part of the annotation-based language's grammar. In this section we will provide a survey of commonly known and recognized structural idioms of annotations' usage. Although for the sake of the discussion about the correspondence a single example would be enough, this overview can be useful for annotations' authors in the process of designing an annotation-based language.

Vectorial annotation idiom described by Guerra et al. [12] enables us to add multiple annotations of the same type to the same target program element. An example showing vector of multiple `@Alternative` annotations is presented in Listing 2. Annotation `@Alternatives` has a single parameter of array of `@Alternative` annotations.

Listing 2. Vectorial annotation idiom example

```
@Alternatives(
   {@Alternative(B.class), @Alternative(C.class)})
public class A { ... }
```

[6]A language focused on, and usually also restricted to a particular problem domain [11].

Composite annotation idiom described by Guerra et al. [12] has annotations as its parameters as well. It allows creation of a tree-like structure of annotations, however, bound to the same target program element. A composite annotation example is shown in Listing 3. In the example there is the `@Author` annotation nesting `@Name` and `@Contact` annotations.

Listing 3. Composite annotation idiom example

```
@Author(
  name=@Name(firstName="Milan", surname="Nosal"),
  contact=@Contact(email="milan.nosal@gmail.com")
)
public class Person { ... }
```

Inside relation idiom requires an annotation @A to be inside scope of another annotation @B. E.g., if @A annotates a field of a class then a class (or its outer class or package) has to be annotated by @B. This idiom was described by Noguera et al. in [1] and also by Ruska et al. in [13], where they call it the Parent-child relation. This idiom is a special case of the Required attribute (annotation) idiom described by Cepa et al. [14] specifying that an annotation requires a usage of another annotation to be valid. In the Required attribute (annotation) idiom the scope is arbitrary. We have already seen an example of the Inside relation idiom in Figure 1 and it is reiterated in Listing 4. In it, the `@Id` and `@Column` annotations had to be used to annotate fields of a class annotated by the `@Entity` annotation. Annotated fields are inside the scope of the `@Entity` annotation annotating the enclosing class.

Listing 4. Inside relation idiom example

```
@Entity(name = "PERSON")
public class Person {

  @Id
  @Column(name = "PER_ID")
  private int id;

  @Column(name = "NAME")
  private String name;
  ...
```

Neighbor idiom specifies that a valid usage of annotation @A is only in case when its target program element is also annotated with another annotation @B. This does not necessarily mean that usage of @B requires @A (this relation is not commutative). This idiom is also a specialization of the Required attribute (annotation) idiom described by Cepa et al. [14]. Noguera et al. [1] describes this idiom as the Requires idiom. A variation of this idiom is described by Ruska et al. in [13] as the Occurrence of multiple annotations idiom that is commutative. In the following example in Listing 5 the `@Table` annotation requires the `@Entity` annotation annotating the same class in order to be considered for processing by the annotations semantics implementation. In this example only an entity class can be mapped to database.

Listing 5. Neighbor idiom example

```
@Entity
@Table("PERSON_TABLE")
public class Person { ... }
```

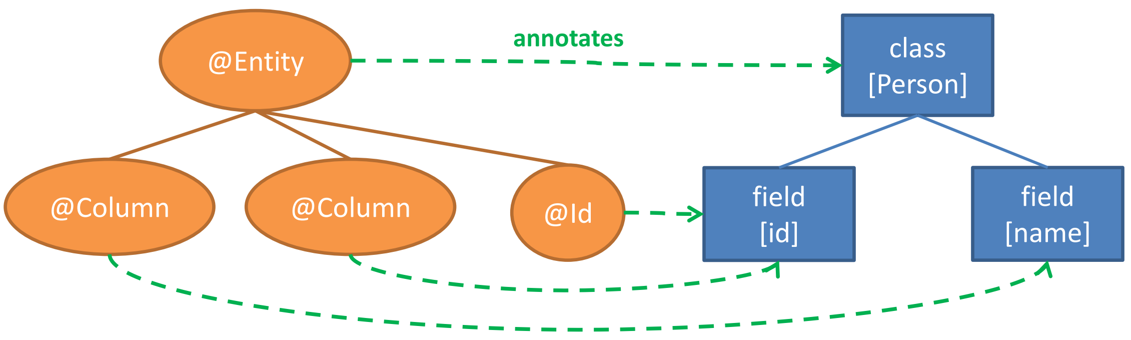


Fig. 2. AST model of the Person entity definition using JPA annotations

Mutual exclusivity idiom prohibits usage of an annotation @A if the target program element is annotated with @B (@A and @B are mutually exclusive). The idiom is described by Ruska et al. in [13]. Noguera et al. [1] call the same idiom the Prohibits idiom. This idiom is a specialization of the Disallowed idiom described by Cepa et al. [14] that can specify exclusivity on various scopes, not only on the same target program element. Following code sample from Listing 6 shows an example of invalid use of annotations that are mutually exclusive. In our annotation-based language a field cannot be annotated both by both `@Id` and `@Basic` because a column cannot be both identifier and regular column at the same time.

Listing 6. Mutual exclusivity idiom violation example

```
// incorrect usage of @Id and @Basic
@Id
@Basic
private String name;
```

Unique annotation occurrence idiom requires that only one instance of an annotation type can be present in a given scope of the host @EL. The idiom is described by Ruska et al. [13]. A variation of the idiom is mentioned by Noguera et al. [1] as the Unique idiom that can be applied to annotation parameters. This way an annotation-based language designer can specify that the parameter value of the annotation can occur only once in a given scope. In the example from Listing 7 there is a violation of the Unique annotation occurrence for `@Id` in the scope of the class. In this language there cannot be two identifiers for the same entity.

Listing 7. Unique annotation occurrence idiom violation example

```
@Table("PERSON")
public class Person {

  @Id
  private int id;

  // cannot be defined again
  @Id
  private String name;

  ...
```

Refers to idiom requires an annotation parameter to be a reference (usually implemented as a simple string) to a parameter value of another annotation. The idiom is described by Noguera et al. [1]. The code snippets in Listing 8 show a Refers to idiom where the `@Author` annotation from class `A` is reused for class `B` by using a reference. `@AuthorRef`'s `value` parameter[7] refers to the `id` parameter of the `@Author` annotation.

Listing 8. Refers to idiom example

```
@Author(
  id="milan.nosal",
  name=@Name(firstName="Milan", surname="Nosal"),
  contact=@Contact(email="milan.nosal@gmail.com")
)
public class A { ... }

@AuthorRef("milan.nosal")
public class B { ... }
```

IV. Annotations' Abstract Syntax Specificities

We discussed an important observation about correspondence between a set of annotations and a formal language in terms of abstract syntax. However, there are also discrepancies between annotations and a common standalone formal language definition. These discrepancies stem from the very nature of annotations. Annotations are meant to annotate program elements of their host @EL. We have already seen it in the example from Figure 1. The `@Entity` annotation annotated the `Person` class. The `@Id` annotation annotated the `id` field. And so on. In Figure 2 these relations based on annotating are represented by dashed arrows from the AST of annotations to the simplified AST of the host @EL. The use of annotations in context of the host language creates an unusual aspect of abstract syntax that is not present in a standalone language – *relations between annotations and host program elements* that are annotated. In general, grammar defines restrictions for relations between concepts of the formal language. However, in case of an annotation-based language, there are also restrictions on relations between annotations (@L concepts) and host language elements (not @L concepts).

[7]If an annotation type declares a `value` parameter then the name of the parameter can be omitted from the annotation instance as in case of `@AuthorRef`.

These restrictions are characteristic for annotations and in language theory they correspond to language composition.

We distinguish two types of relations between annotations and host language program elements, based on the relation direction:

- *code-wise relations* define annotations' requirements posed on their target elements and the program they are used in, and
- *reversed code-wise relations* specify host language's requirements for the annotations to be used in a context of a given host program element.

In the following sections we will discuss in detail both types.

A. Code-wise Relations

Code-wise relations are relations defining restrictions by annotations to their target program elements. Each annotation may specify some requirements for its target program element. These annotations cannot annotate arbitrary host language element, because the given restrictions have to be kept. Basically, code-wise relations specify on which program elements annotations can be used.

As a standard example of enforcing existing code-wise relations we can mention Java's `@Target` metaannotation (annotation that annotates annotation types). Using the `@Target` metaannotation we can specify what types of Java program elements can a given annotation annotate. E.g., we can use `@Target(ElementType.METHOD)` to restrict annotations to annotate only methods. In C# the support for this type of restrictions is even of a finer grain. However, there are still many useful restrictions that cannot be defined by standard tools. E.g., we cannot restrict an annotation to annotate only an implementation of some interface, such as restricting the `@WebServlet` annotation to annotate only an implementation of the `Servlet` interface.

To illustrate it on a real world example, we can take a look back at the JPA annotations. The code snippet in Listing 9 replays the Person entity mapping with an added field for the person's age. However, JPA does not allow mixing annotating fields and getters/setters. In this case, the last `@Column` annotation will not be processed by the JPA implementation. In case of the `age` field, the JPA `@Column` annotation annotates the setter method, not like in the case of the other fields, which violates JPA specification (we could say it violates the JPA annotations abstract syntax).

Listing 9. Incorrect JPA mapping
```
@Entity(name = "PERSON")
public class Person {
  // an annotated field - OK
  @Id
  @Column(name = "PER_ID")
  private int id;

  // another annotated field - OK
  @Column(name = "NAME")
  private String name;

  private int age;

  // an annotated setter - incorrect
  // (it should annotate the field)
  @Column(name = "FIRST_NAME")
  public void setAge(int age) { ... }
  ...
```

B. Idioms in Code-wise Relations

There are common structure stereotypes in code-wise relations as well. They indicate that this aspect of annotations usage structure has to be considered a part of annotations abstract syntax. Again, listed idioms are not only an evidence of the discussed aspect of the abstract syntax, but also can be inspirational for annotations' authors. Following is a list of known code-wise relations idioms.

Target restriction idiom specifies which types of target program elements can be annotated by the annotations. This idiom is implemented both in Java and C# as a standard validation. Noguera et al. [1] introduces finer-grained validations of the Target restriction idiom on Java platform that they call AvalTarget. Kellens et al. [15] focus on designing a framework for declaring arbitrary requirements on annotations' target program elements. Code sample in Listing 10 shows a standard Target restriction implementation in Java. The `@Override` annotation can annotate only methods, because only methods can override a behaviour. In addition we would want to restrict `@Override` to methods that really override inherited methods. In Java this checking is implemented in standard compiler for the `@Override` annotation, but in general checking like this is left to the annotations author.

Listing 10. Target restriction idiom example
```
// compile-time error
@Override
public class Clazz {

  // valid use (toString overrides inherited method)
  @Override
  public String toString() { ... }
}
```

Refers to element idiom specifies that an annotation parameter is a reference to program element from @EL (other than annotations target program element). The idiom is described by Ruska et al. in [13] as Annotation values referencing other elements idiom. A specialized version of the idiom where the annotation refers to a class is described by Guerra et al. [12] as an Associative Annotation idiom. Code sample from Listing 11 shows a usage of the Refers to element idiom to refer to a specific validation implementation class for an annotation type through a parameter of the `@Validator` metaannotation. The `Validation` interface defines validation operation for a given annotation annotated by the `@Validator` metaannotation. The concrete `OverrideValidation` defines specific validation for the `@Override` annotation.

Listing 11. Refers to element idiom example
```
public @interface Validator {
  Class<? extends Validation> value();
}
...
@Validator(OverrideValidation.class)
public @interface Override { }
```

Type idiom requires that target program element yields or conforms to a specific runtime type in host @EL. An example might be the type of the field, or the interface of the class. This idiom is described by Noguera et al. [1]. One of the uses for the Type idiom is using annotations as type modifiers (e.g., `@NotNull` annotating only reference types). The following code in Listing 12 shows a valid use of `@WebServlet`. The `@WebServlet` annotation registers a servlet to the web container. Therefore the annotation is supposed to annotate only an implementation of the `Servlet` interface.

Listing 12. Valid Type idiom example

```
@WebServlet(urlMappings={"/MyApp"})
public class MyAppServlet implements Servlet {
  ...
```

C. Reversed Code-wise Relations

Reversed code-wise relations pose restrictions on the host language according to the annotations that are present in the code. E.g., the same way as we can specify the type of the field of a class is the `Person` class, we could also require that the field type (in this case the `Person` class) has to be annotated by a particular annotation (e.g., `@Entity`). Or, we may want an argument of a method to be of a type that is annotated by a particular annotation. Basically, reversed code-wise relations specify how can host language use annotated program elements.

Such requirements we call reversed code-wise relations because they revert the direction that we use to look at annotations and their binding with the host language. Code-wise relations specify in which cases the annotations are not correctly used. Reversed code-wise relations specify in which cases the host language source code is not correctly used with respect to annotations. According to code-wise relations the annotation is invalid if its target program element (or some other program element of the host language) does not exhibit some characteristics. According to reversed code-wise relations the program element is incorrectly used if it or some related program element misses a particular annotation (or has an incorrect one). This kind of relations increases integration of annotations into the host language.

As an example of an error caused by violating this type of requirements is a quite common error in using dependency injection in Enterprise Java. Code sample in Listing 13 shows an invalid use of dependency injection. Both classes are marked as stateless beans and therefore should be managed by a container. `@EJB` annotation marks a server of a mail service implementation to be injected. However, then the `Notifications` class takes control of the `MailService` instantiation by using the `new` keyword. When the objects are created like this, the dependency injection annotations will not take effect and the server will not be injected. This is a case of reversed code-wise relation, because the annotations are used properly, however, the code that uses annotated program elements is not correct. The `MailService` instance should be obtained through the `InitialContext` to be correct.

Listing 13. Using `@EJB` for dependency injection

```
@Stateless
public class MailService {

  @EJB
  private Server server;

  public void sendMail(String mail) {
    server.send(mail);
  }

  ...
}

@Stateless
public class Notifications {
  public void notify() {
    MailService ms = new MailService();
    ms.sendMail(
       "This will throw NullPointerException.");
  }
}
```

D. Idioms in Reversed Code-wise Relations

We have recognized and identified following two reversed code-wise validations:

Annotated type idiom poses a restriction upon entities[8] of the host @EL. It requires that an entity type has to be annotated by an annotation. E.g., a method argument might require an object of a type annotated with a specific annotation. This idiom can enhance the object-oriented dynamic binding of the @EL. Using this idiom a method could require object of any type but annotated with a particular annotation, or with particular annotations in its scope. In the code sample from Listing 14 we use the `serialize` method to serialize an object of the `Person` class. However, using the Annotated type idiom the method serialize requires that the type of the object to be serialized to be annotated by the `@Serializable` annotation (e.g., instead of Serialize marker interface). Therefore the `Person` class has to be annotated by the `@Serializable` annotation for the example to be valid.

Listing 14. Annotated type idiom example

```
public class Serializer {
  // serialize requires argument with
  // type annotated by @Serializable
  public static void serialize(
      @AnnotatedType(Serializable.class)
      Object object) { ... }
}

...

public static void main(String[] args) {
  Person person = new Person("Milan");
  Serializer.serialize(person);
}

...

// for program to be valid, the Person
// class has to be annotated with @Serializable
@Serializable
public class Person { ... }
```

[8]Method argument, variable, etc.

Annotated program element idiom requires that the program elements which exhibit some particular characteristics must be annotated by a particular annotation. This idiom is very close to the work of Kellens et al. [15] that uses dependencies like these for co-evolution of annotations with source code. As an example we can use the `@Override` Java annotation in Listing 15. If a method is overriding a method from a superclass it has to be annotated by the `@Override` annotation. If the annotation is not used the program should be incorrect (as in case of the `override` keyword in C#). Code in Listing 15 shows examples of both valid and invalid program elements.

Listing 15. Annotated program element idiom example

```
public class A {
  public void a() {}
  public void b() {}
}

public class B extends A {

  // invalid program element
  public void a() {}

  // valid program element
  @Override
  public void b() {}
}
```

V. Abstract Syntax Correspondence Summary

According to the presented discussion we can summarize our conclusions as follows. Considering the abstract syntax (AS) of a language defined by source code annotations we have to consider following components of annotations' AS:

- relations between annotations – **structural restrictions**,
- annotations' requirements on @EL concepts – **code-wise restrictions**, and
- @EL concepts' requirements on annotations – **reversed code-wise restrictions**.

While the relations between annotations match the standard abstract syntax of a standalone formal language, the other two are specific for annotations. The code-wise and reverse code-wise restrictions can be in general compared to embedding as a form of language composition, but annotations tend to come with more complex constraints on their usage and in case of an embedded language the embedded portions are usually not dependent on each other (without structural restrictions). And, from the implementation viewpoint, embedded languages usually have own parser.

If we consider a set of related annotations an *annotation-based language*, then based on our discussion we will define the abstract syntax of an @L as in Definition 2. So far there is not a standard formal apparatus for describing the full AS of @L. However, there are several approaches to this problem in the academy that we will review in section IX.

Definition 2. *The abstract syntax of an @L is a set of structural, code-wise and reversed code-wise restrictions on @L annotations' usage. These restrictions represent formation rules that specify a valid @L sentence.*

VI. Concrete Syntax Correspondence

Of course, the correlation between annotations and formal languages does not end with abstract syntax aspect of the formal language description. Clearly, the annotations themselves need to have a concrete way of presentation (and serialization). In this section we will take a look at how the concrete syntax of annotations is defined.

@L author can design the @L in terms of the abstract syntax as we have already discussed. Of course, it is only logical that she should be also able to specify how the annotations of the @L will be presented in the source code of the host language. In most common @EL the apparatus for the concrete syntax definition are *annotations type*s. A specific of annotations in terms of concrete syntax when compared to formal languages is annotations' binding to target program elements. Both of these aspects will be discussed in following sections.

A. Annotation Types

Annotations are instances of annotation types the same way as objects are instances of classes in object-oriented programming. An annotation type is a definition of a structure of a set of annotations. It defines what parameters can an annotation have, what is the name of the annotation, etc. Using annotation types the @L author can specify what are the @L terminal symbols – annotations' and parameters' names, and their values' types. Just for illustration we can take a look at the code snippet in Listing 16 presenting a simple annotation in C#. This annotation type defines annotations with name "Configuration" and two parameters, the first an integer identified as "paramId", and thesecond a string identified as "paramValue". If the @L consisted only of annotations of this type, we could easily identify its lexical symbols - "Configuration", "paramId", "paramValue", integer value and a string. The Java version of the same annotation type is presented in Listing 17.

Listing 16. C# version of the `Configuration` annotation type

```
public class Configuration : System.Attribute
{
  public int paramId;
  public string paramValue;
}
```

Listing 17. Java version of the `Configuration` annotation type

```
public @interface Configuration
{
  public int paramId();
  public string paramValue();
}
```

Another interesting fact the reader might have noticed is that the annotation types can partially define abstract syntax. If one of the parameters accepted annotations[9], then the annotation would be an instance of the Composite annotation idiom.

However, even in CS there are discrepancies between annotations and formal languages. The CS of annotations is restricted to follow some rules in order to be parsable by the

[9]Currently, annotation nesting is not supported in C#, but there are implementations that allow it, e.g., Java annotations.

standard @EL parser (or the 3rd party tool). In @EL GPLs those rules are defined by the GPL grammar.

Let us take a look at a set of grammar rules[10] of Java 8 that specify the grammar for Java annotations that are listed in Listing 18.

Listing 18. Grammar rules for the Java 8 annotations

```
Annotation ->
   NormalAnnotation | MarkerAnnotation
   | SingleElementAnnotation

NormalAnnotation ->
   '@' TypeName '(' [ElementValuePairList] ')'

ElementValuePairList ->
   ElementValuePair {',' ElementValuePair}

ElementValuePair -> Identifier '=' ElementValue

ElementValue ->
   ConditionalExpression
   | ElementValueArrayInitializer | Annotation

ElementValueArrayInitializer ->
   { [ElementValueList] [','] }

ElementValueList ->
   ElementValue {',' ElementValue}

MarkerAnnotation -> '@' TypeName

SingleElementAnnotation ->
   '@' TypeName '(' ElementValue ')'
```

These grammar rules specify CS restrictions on Java annotations. As we can see, annotations have to start with the '@' sign, followed by the annotation name. Then there are optional annotation parameters enclosed in parentheses, and so on. Considering we had an Java annotation type for the `Configuration` annotation type presented in Listing reflst:javaAnnType, then a concrete Java annotation would look like the annotation in Listing 19.

Listing 19. Java `@Configuration` annotation example

```
@Configuration(paramId=1, paramValue="new")
public class A {
   ...
```

Of course, the different attribute-oriented programming implementations might have different restrictions. E.g., in C# the annotations are not prefixed by the '@' but enclosed with brackets, and have some other minor differences from Java annotations. The same `Configuration` annotation in C# (annotation type in Listing 16) would look like the code in Listing 20.

Listing 20. C# version of the `Configuration` annotation

```
[Configuration(paramId=1, paramValue="new")]
public class A {
   ...
```

In a conventional formal language the language author is usually not restricted by any such rules. In this aspect annotations resemble more generic languages[11], such as XML languages, and alike; that are built around a given syntactic skeleton.

B. Binding Rules

@L CS has another specific aspect – *binding rules*. Each annotation *marks* (annotates) its target program element. This relationship is expressed by the relative position of annotation to its target element. Again, the binding rules for a specific @OP implementations might differ. These rules have to ensure that for each annotation there will be unambiguous mapping to its target language element and that for each program element there will be unambiguous way of finding its annotations. The most common binding for annotations is using annotations as prefixes[12]. E.g., the Java annotations are considered modifiers and therefore they prefix declarations just as Java modifiers do. The reader has seen already seen multiple examples of Java annotations usages. This can be also illustrated by Java grammar for class modifiers in the Java 8 grammar excerpt[13] in Listing 21. The excerpt shows that class annotations prefix the class declarations.

Listing 21. Grammar excerpt for class modifiers showing in-place binding of Java 8 annotations

```
NormalClassDeclaration ->
   {ClassModifier} 'class' Identifier ...

ClassModifier ->
   Annotation | 'public' | ...
```

Since Java 8, annotations are supported on types, too. Listing 22 is a grammar rule[14] for primitive types that shows type annotations allowing to annotate types.

Listing 22. Grammar rule illustrating type annotations

```
PrimitiveType ->
   {Annotation} NumericType
   | {Annotation} 'boolean'
```

There might be additional rules for binding, as for example a restriction in Java requiring that two annotations of the same type cannot annotate the same target program element.

C. Summary

Each particular @EL defines its own concrete syntax skeleton for adding annotations to its source code. While the restrictions posed by @EL have to be kept, the @L author can use annotation types to define the rest of the concrete syntax. Therefore we will define the concrete syntax aspect of @L by Definition 3.

Definition 3. *The concrete syntax of an @L is specified by restrictions posed by the host @EL in combination with the*

[10]Source: http://docs.oracle.com/javase/specs/jls/se8/html/jls-19.html

[11]Mernik [16] calls a generic language Commercial Off-The-Shelf (COTS). We find the *generic language* term by Chodarev et al. in [17] more intuitive and therefore we will keep using this term.

[12]However, in general, annotations do not have to prefix annotated program elements. It is sufficient to have an unambiguous rule of how to relate annotations with their target language elements.

[13]The ellipsis (`...`) indicates that we have shortened the rules to show just the relevant parts. Source: http://docs.oracle.com/javase/specs/jls/se8/html/jls-19.html

[14]Source: http://docs.oracle.com/javase/specs/jls/se8/html/jls-19.html

set of concrete annotation types of annotations that belong to the @L.

VII. Semantics Correspondence

According to Kleppe [18], a sound language description would not be complete without a semantics description. And of course, the same applies to annotation-based language. There are multiple ways of describing semantics of a language. An annotation-based language is usually described by dynamic semantics that is defined by the tool processing the annotations (the *reference implementation* [18]).

There are two approaches to @L reference implementation:

- **compile time processing** is implemented as a pluggable annotation processor plugged to the host @EL compiler, and
- **runtime processing** is implemented as a reflection mechanism.

Compile time processing is implemented as a pluggable annotation processor. The host @EL parser creates an AST with annotations and provides it to all registered annotation processors. The AST with annotations can be used to generate code or other software artefacts, to generate documentation or even to manipulate the AST. E.g., in Java there is a standard implementation of pluggable annotation processing API released under JSR 269 specification[15]. This standard annotation processing API does not support AST modification and can be only used to generate new artefacts. An alternative to JSR 269 is a Spoon API by Pawlak [19] that enables fine grained source code modifications.

Runtime processing is implemented as an API that allows some form of runtime reflection for querying annotations. Runtime processing is usually used to read configuration of frameworks and programs. Languages such as Java or C# provide standard Reflection API that can be used to query for annotations on program elements such as classes, method, etc. These can be used to find out whether there is a particular annotation annotating the chosen program element. However, these APIs do not enable searching for annotations in a set of program elements (e.g. finding all the occurrences of specific annotation on all the classes on classpath) and likewise operations. These types of queries common in compile-time annotation processors. The need for the same feature in runtime is reflected by commercial runtime APIs, such as Scannotation[16] or Google Reflections[17].

An interesting hybrid of the compile-time and runtime processing is aspect-oriented programming (AOP) with annotations. Annotations in AOP can be used to bind aspects to program elements. This way we can add, remove, or modify the code. The final weaving of the aspect may happen both during compile-time and runtime, depending on used AOP implementation.

[15]https://www.jcp.org/en/jsr/detail?id=269
[16]http://scannotation.sourceforge.net/
[17]https://github.com/ronmamo/reflections

Each annotation-based language defines its semantics using one of the discussed approaches. Thus, we define @L operational semantics by Definition 4.

Definition 4. *The @L semantics is described by reference implementation using a pluggable annotation processor or a GPL code using reflection API. Reference implementation may use convenience frameworks, such as Google Reflections.*

VIII. Annotation-based Language

We presented our observations indicating a close correspondence between source code annotations and formal languages. We proposed definitions of the three main annotation-based language definition components - @L abstract syntax, concrete syntax and semantics. Based on the presented discussion we propose to define a term *annotation-based language* to describe a given set of annotations that are processed by the same reference implementation with the same goal. For example, if we have a set of JPA annotations used to describe mapping of Java classes to relational database that are processed by a JPA implementation (e.g., Hibernate), we can consider them an @L. Our formulation of the @L definition is presented in Definition 5. In it we assume that the same reference implementation implies the same annotations problem domain (e.g., object-relational mapping).

Definition 5. *Annotation-based language (@L) is a set of all annotations and their parameters (*alphabet*) processed by the same reference implementation. It is defined by the reference implementation (*semantics*), structural, code-wise and reverse code-wise restrictions (*grammar – abstract syntax*), and their annotation types (*grammar – concrete syntax*).*

For example, if we have a set of JPA annotations used to describe mapping of Java classes to relational database that are processed by a JPA implementation (e.g., Hibernate), we can consider them an @L. This @L is of course defined by all three language aspects of CS, AS and semantics according to definitions we have proposed.

We have also noticed that the main source of the discrepancies between @L and a conventional formal language is the binding of annotations to target elements of the host language. Therefore we will emphasize the importance of the binding in the @L. We can therefore identify two main components of the @L:

- @L **concepts** represented by annotations and their structural relations, and a
- *meaningful* **binding** between concepts from @L and host @EL (represented by code-wise and reverse code-wise relations).

These two @L components are illustrated on the example with JPA mapping of the Person class in Figure 3. @L concepts are the annotations themselves, the binding maps the annotations to host language program elements.

We expect the binding between annotations and their target elements to be meaningful. That means that changing the target element of an annotation should also change the meaning of

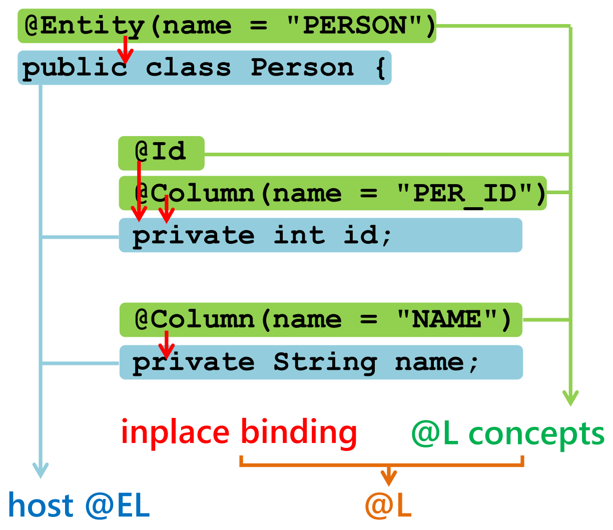


Fig. 3. Illustration of concepts and binding @L components on JPA example

the @L sentence to which the annotation belongs. E.g., if we consider the example of JPA configuration in Listing 4, moving the `@Id` annotation from the `id` field to the `name` field would define a different object-relational mapping, although the @L sentence would consist of the same annotations.

IX. Related Work

There are several works that aim at annotations' relations definition[18]. They either provide a framework to define and validate more-or-less arbitrary dependencies between annotations and their target elements or they provide implemented common stereotypes in @L structure that can be reused for validation. These idioms show common stereotypes in annotation-based abstract syntax definition. Most of the works were already referred throughout the paper, in this section we will provide a brief summary.

Darwin [21] devised a DSL for dependencies description. His aim, contrary to the other works, was to provide a framework for describing dependencies for third party @Ls. The author of rules was to be a user of @L and not its author. Ruska et al. [13] identify several structural idioms and provide a framework for checking dependencies. In their approach they store the source code model into a database and run SQL queries representing constraints. They designed a Prolog-like DSL for convenient rules writing. Although both of these works do not explicitly mention the term annotation-based language (or any synonym) they admit that annotations have some structured dependencies that we consider a recognition of abstract syntax of @L.

Kellens et al. [15] introduce so called Smart Annotations. Smart Annotations declare their sufficient and necessary requirements. Necessary requirements declare what characteristics the annotated element must exhibit so the annotation usage is valid. Sufficient requirements are dual to necessary and they declare that if there is an entity in the code exhibiting characteristics required by it then it should be annotated by the annotation. Their work is focused on code-wise dependencies and they aim to provide better control of evolution of annotated software. Their sufficient requirements exhibit characteristics of the reversed code-wise dependencies.

Cepa et al. [14] is one of the first works that concerns checking dependencies between annotations and the host @EL. Although their framework was quite limited in supported restrictions[19], in their work they also look at annotations as a form of a language framework. They realized that annotations can be used to enhance the host GPL with domain-specific concepts thus acknowledging @OP as a form for DSL-like language extensions.

Noguera et al. [1] went even further than Cepa et al. and stated that *"a set of annotations dedicated to a given domain-specific concern can be referred to as an Attribute Domain-Specific Language (AttDSL)"*. They therefore consider annotations a language and not only a language extension. They distinguished between structural and code-wise dependencies (we used their naming).

Song et al. [22] introduce metadata invariants that are not exclusive for annotations but work for XML languages, too. They use Metadata Invariant Language (MIL) to write invariants for annotations or XML documents. MIL can be used to check structural and code-wise dependencies. Their work assumes that for an @L there can be a standalone external language that shares part of the @L semantics (multiple concrete syntaxes). We have utilized the same property of a subset of @Ls in our previous work [23]. We designed a tool that is able to create AST for @L and combine it with concepts from a separate XML-based language that is an alternative notation for the @L. The tool takes a mapping between the two formats.

The furthest advance towards @L we consider the work of Noguera and Duchien [24]. Not only they recognize @L, but they also designed a method for creating Annotation Model, a UML model of annotation-based language. They use this model and its binding to code model (AST of the host @EL) to check annotations restrictions.

Cepa in his book about @OP [5] connects the annotations to languages too. He mentions that annotations can be seen as a convenient way of extending the grammar of the language. With the work of Noguera et al. [1] Cepa's ideas were the main inspiration for this work. However, in his work he considers the annotations only as an extension of the grammar and as an alternative for adding domain-specific abstractions to the source code, and only briefly mentions annotations as a restricted form of embedded DSL. He does not further examine the correspondence between annotations and languages.

[18]We focus solely on the source code annotations defined in section II as a language implementation strategy. Therefore we will not discuss works such as Bonenfant et al. [20], which also use the term *annotation language*. However, they use it in different context than us. In their work the annotations are any custom metadata (their *annotation language* is an XML-based language).

[19]E.g., it cannot check dependencies between annotations on program elements in the same scope in program model.

From the viewpoint of existing @Ls, we could find multiple sets of annotations defined for the same domain and processed by the same semantics reference implementation. There are sets of annotations defining GUI from data model [25], annotations used for plugin extension definition [26], @L for design patterns definition [27], @L for model checking [28], and so on.

There are not many works concerning @L design, however, one can find some inspiration in analysis provided by Mancini et al. [29]. They discuss options of using annotations and their design for data validation definition. All the above mentioned works about annotations usage restrictions define some annotations AS idioms that can be used for @L design as well. Guerra et al. [12] discuss solely annotations idioms both in CS and AS. Correia et al. [6] discuss bad smells that can be result of bad annotations design or their usage. They also provide a set of possible solutions to remove them. Correia et al. [6] show that annotations syntax can be as important as their domain usability [30].

X. Conclusion

This paper presented our observations about correspondence between source code annotations and formal languages. The correspondence was illustrated on three definition aspects of formal languages – abstract syntax, concrete syntax and semantics. For each of these aspects also the discrepancies were discussed. While the correspondence indicates that we should look at annotations as a form of language (and therefore we define the term *annotation-based language* for a set of domain-specific annotations), the discrepancies identify the main specificity of annotations that separates it from conventional formal languages. This specificity is the binding of annotations to their host annotation-enabled language.

Realizing the correspondence between annotations and formal languages we can draw some consequences for future research directions. For example, XML generic language provides several mechanisms to define an abstract and concrete syntaxes of concrete XML languages, e.g., XML schema definition, Document Type Definition, etc. In practice we can notice notorious lack of similar mechanisms for @Ls. Annotation types are usually not sufficient to describe the relations that are common in @L. A common practice is to describe the grammar using natural language documentation. Considering the presented correspondence, then instead of natural language documentation, the formal methods of abstract syntax definition can be applied. Methods such as EBNF are commonly known and therefore their application can make the annotation-based language syntax more comprehensible. In section IX we discussed several academic works that implement frameworks for @L abstract syntax definition, but so far none of them became industrial standard. So a logical consequence of the correspondence is the *need for standard AS definition formalism/mechanism for annotations*. In this matter there was a small step further in Java 8 type annotations.

The standard AS definition formalism/mechanism for annotations cannot be restricted to mere @L AS validation. Another important feature is the ability to create a fully annotated AST. In Figure 2 we have sketched the AST with annotations' relations in mind. However, currently the @EL implementations do not support @L AS and therefore the AST nodes representing annotations have no explicit relations with other annotations (unless annotation types support it, such as in case of Composition annotation idiom in Java). Supporting these relations explicitly could prove beneficial for @L authors and their semantics implementation.

Another observation that is related to our discussion is the lack of unified API for semantics reference implementation in runtime and in compile time. Currently, standard @EL implementations provide two distinct APIs for annotation processing and for runtime reflection. This observation was authored by Cepa [5] quite a long time ago, but to our best knowledge no real advance was made in industry in this matter.

In general we can consider annotations a *generic embedded language*. They provide a syntactic skeleton that on one hand restricts @L author in concrete and abstract syntax, but on the other hand provides standard tools for their parsing (either the host language parser or a third party tool). But in contrast to generic languages they are restricted to embedding into the host language; they have to annotate its program elements. Based on this observation, in our future work we want to analyze options of using annotations for language composition. From the observations presented in section IV we learned that annotations' specificity is their binding to host language. In the future work we want to examine types of language composition that can utilize annotations as an implementation technique.

References

[1] C. Noguera and R. Pawlak, "AVal: an extensible attribute-oriented programming validator for Java: Research Articles," *Journal of Software Maintenance and Evolution*, vol. 19, no. 4, pp. 253–275, Jul. 2007. http://dx.doi.org/10.1002/smr.349

[2] R. Rouvoy and P. Merle, "Leveraging Component-Oriented Programming with Attribute-Oriented Programming," in *Proceedings of the 11th International ECOOP Workshop on Component-Oriented Programming*, ser. WCOP 2006, 2006.

[3] S. Chodarev, D. Lakatoš, J. Porubän, and J. Kollár, "Abstract syntax driven approach for language composition," *Central European Journal of Computer Science*, vol. 4, no. 3, pp. 107–117, 2014. http://dx.doi.org/10.2478/s13537-014-0211-8

[4] D. Lakatoš, J. Porubän, and M. Bačíková, "Declarative specification of references in DSLs," in *2013 Federated Conference on Computer Science and Information Systems*, ser. FedCSIS 2013, Sept 2013, pp. 1527–1534.

[5] V. Cepa, *Attribute enabled software development: illustrated with mobile software applications*. Saarbrücken, Germany: VDM Verlag, 2007.

[6] D. A. A. Correia, E. M. Guerra, F. F. Silveira, and C. T. Fernandes, "Quality Improvement in Annotated Code," *CLEI Electron. J.*, vol. 13, no. 2, 2010, article ID 7.

[7] M. Nosáľ and J. Porubän, "XML to Annotations Mapping Definition with Patterns," *Computer Science and Information Systems*, vol. 11, no. 4, pp. 1455–1477, 2014. http://dx.doi.org/10.2298/CSIS130920049N

[8] J. Kollár, I. Halupka, S. Chodarev, and E. Pietriková, "pLERO: Language for grammar refactoring patterns," in *2013 Federated Conference on Computer Science and Information Systems*, ser. FedCSIS 2013, Sept 2013, pp. 1503–1510.

[9] C. Noguera, A. Kellens, D. Deridder, and T. D'Hondt, "Tackling Pointcut Fragility with Dynamic Annotations," in *Proceedings of the 7th Workshop on Reflection, AOP and Meta-Data for Software*

Evolution, ser. RAM-SE '10. New York, NY, USA: ACM, 2010, pp. 1:1–1:6. http://dx.doi.org/10.1145/1890683.1890684

[10] W. Cazzola and E. Vacchi, "@Java: Bringing a richer annotation model to Java," *Computer Languages, Systems & Structures*, vol. 40, no. 1, pp. 2–18, 2014, special issue on the Programming Languages track at the 28th ACM Symposium on Applied Computing. http://dx.doi.org/10.1016/j.cl.2014.02.002

[11] S. Zawoad, M. Mernik, and R. Hasan, "FAL: A forensics aware language for secure logging," in *2013 Federated Conference on Computer Science and Information Systems*, ser. FedCSIS 2013, Sept 2013, pp. 1579–1586.

[12] E. Guerra, M. Cardoso, J. Silva, and C. Fernandes, "Idioms for Code Annotations in the Java Language," in *Proceedings of the 17th Latin-American Conference on Pattern Languages of Programs*, ser. Sugar-LoafPLoP, 2010, pp. 1–14.

[13] Š. Ruska and J. Porubän, "Defining Annotation Constraints in Attribute Oriented Programming," *Acta Electrotechnica et Informatica*, vol. 10, no. 4, pp. 89–93, 2010.

[14] V. Cepa and M. Mezini, "Declaring and Enforcing Dependencies Between .NET Custom Attributes," in *Generative Programming and Component Engineering*, ser. Lecture Notes in Computer Science, G. Karsai and E. Visser, Eds. Springer Berlin Heidelberg, 2004, vol. 3286, pp. 283–297. http://dx.doi.org/10.1007/978-3-540-30175-2_15

[15] A. Kellens, C. Noguera, K. De Schutter, C. De Roover, and T. D'Hondt, "Co-evolving Annotations and Source Code through Smart Annotations," in *14th European Conference on Software Maintenance and Reengineering*, ser. CSMR 2010, 2010, pp. 117–126. http://dx.doi.org/10.1109/CSMR.2010.20

[16] M. Mernik, "An object-oriented approach to language compositions for software language engineering," *Journal of Systems and Software*, vol. 86, no. 9, pp. 2451–2464, 2013. http://dx.doi.org/10.1016/j.jss.2013.04.087

[17] S. Chodarev and J. Kollár, "Language Development Based on the Extensible Host Language," in *Proceedings of CSE 2012 International Scientific Conference on Computer Science and Engineering*. EQUILIBRIA, s.r.o., 2012, pp. 55–62.

[18] A. Kleppe, "A Language Description is More than a Metamodel," in *4th International Workshop on Language Engineering*, ser. ATEM 2007, 2007.

[19] R. Pawlak, "Spoon: Compile-time Annotation Processing for Middleware," *IEEE Distributed Systems Online*, vol. 7, no. 11, pp. 1–, Nov. 2006. http://dx.doi.org/10.1109/MDSO.2006.67

[20] A. Bonenfant, H. Cassé, M. de Michiel, J. Knoop, L. Kovács, and J. Zwirchmayr, "FFX: A Portable WCET Annotation Language," in *Proceedings of the 20th International Conference on Real-Time and Network Systems*, ser. RTNS '12. New York, NY, USA: ACM, 2012, pp. 91–100. http://dx.doi.org/10.1145/2392987.2392999

[21] I. Darwin, "AnnaBot: A Static Verifier for Java Annotation Usage," *Advances in Software Engineering*, vol. 2010, p. 7, 2010, article ID 540547. http://dx.doi.org/10.1155/2010/540547

[22] M. Song and E. Tilevich, "Metadata invariants: checking and inferring metadata coding conventions," in *Proceedings of the 2012 International Conference on Software Engineering*, ser. ICSE 2012. Piscataway, NJ, USA: IEEE Press, 2012, pp. 694–704. http://dx.doi.org/10.1109/ICSE.2012.6227148

[23] M. Nosáľ and J. Porubän, "Supporting multiple configuration sources using abstraction," *Central European Journal of Computer Science*, vol. 2, no. 3, pp. 283–299, Oct. 2012. http://dx.doi.org/10.2478/s13537-012-0015-7

[24] C. Noguera and L. Duchien, "Annotation Framework Validation Using Domain Models," in *Proceedings of the 4th European Conference on Model Driven Architecture: Foundations and Applications*, ser. ECMDA-FA '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 48–62. http://dx.doi.org/10.1007/978-3-540-69100-6_4

[25] M. Monteiro, P. Oliveira, and R. Goncalves, "GUI generation based on language extensions: a model to generate GUI, based on source code with custom attributes," in *Proceedings of the 10th International Conference on Enterprise Information Systems*, ser. ICEIS 2008, Jun. 2008, pp. 449–452. http://dx.doi.org/10400.8/147

[26] R. Wolfinger, M. Löberbauer, M. Jahn, and H. Mössenböck, "Adding genericity to a plug-in framework," *SIGPLAN Not.*, vol. 46, no. 2, pp. 93–102, Oct. 2010. http://dx.doi.org/10.1145/1942788.1868308

[27] P. Kajsa and P. Návrat, "Design Pattern Support Based on the Source Code Annotations and Feature Models," in *SOFSEM 2012: Theory and Practice of Computer Science*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, vol. 7147, pp. 467–478. http://dx.doi.org/10.1007/978-3-642-27660-6_38

[28] G. Ferreira, E. Loureiro, and E. Oliveira, "A Java Code Annotation Approach for Model Checking Software Systems," in *Proceedings of the 2007 ACM Symposium on Applied Computing*, ser. SAC '07. New York, NY, USA: ACM, 2007, pp. 1536–1537. http://dx.doi.org/10.1145/1244002.1244330

[29] F. Mancini, D. Hovland, and K. Mughal, "Investigating the Limitations of Java Annotations for Input Validation," in *Proceedings of International Conference on Availability, Reliability, and Security, 2010*, ser. ARES '10, Feb 2010, pp. 513–518. http://dx.doi.org/10.1109/ARES.2010.29

[30] M. Bačíková and J. Porubän, "Domain usability, user's perception," in *Human-Computer Systems Interaction: Backgrounds and Applications 3*, ser. Advances in Intelligent Systems and Computing. Springer International Publishing, 2014, vol. 300, pp. 15–26. http://dx.doi.org/10.1007/978-3-319-08491-6_2