Computer Science and Information Systems 13(3):707–729 DOI: 10.2298/CSIS160114024N

Language Composition Using Source Code Annotations

Milan Nosáľ, Matúš Sulír, and Ján Juhár

Technical University of Košice, Department of Computers and Informatics
Letná 9, Košice, Slovak Republic
milan.nosal@gmail.com, {matus.sulir, jan.juhar}@tuke.sk

Abstract. In this paper we examine source code annotations from the viewpoint of formal languages – we discuss their abstract syntax, concrete syntax, and semantics, thus showing the correspondence between annotations and formal languages. We propose to consider a set of all annotations and their parameters processed by the same reference implementation (they belong to the same domain) to be called an *annotation-based language*. The performed analysis also pinpoints a specificity of annotations in comparison with formal languages in general – the binding between annotations and a host language. We elaborate this idea with an analysis of annotations' potential for language composition, in particular for pure embedding. We then show how pure embedding with annotations can be used for language unification, language referencing by extension, and language extension. This work provides a basis for further research in the field of source code annotations in the context of formal languages.

Keywords: source code annotations, annotation-based language, language composition, annotation-based language composition, pure embedding.

1. Introduction

Attribute-oriented programming (abbreviated: @OP) as a technique of marking source code elements with *source code annotations* [30] 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 [4] and references definition [26].

In our previous work [31] we analysed 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 the original version of this paper [32] presented at the FedCSIS conference we examined the relationship of annotations and formal languages in general. We hypothesized that *there is a correspondence between annotations and conventional formal languages* and we have supported the hypothesis by a throughout analysis of annotations from the aspects of concrete syntax (CS), abstract syntax (AS), and semantics. A formal language is defined by an alphabet and a set of formation rules (a grammar [7]). We have shown that the same can be applied to annotations. This is due to annotations' resemblance to domain-specific

languages (abbreviated: DSL, a domain-specific language is a language tailored for a specific, rather narrow domain [23, 28][1]).

In this extended version of the paper we will address the following topics. First, we will provide a brief summary of the work presented in the original paper. We will review the correspondence between annotations and formal languages (DSLs in particular), discrepancies between them, and finally we will provide a definition of the annotation-based language. We refer the reader to the conference version of the paper [32] for the more detailed analysis. In this extended version we elaborate on the idea that stemmed from the observed correspondence. We noticed that we can look at source code annotations as a *generic purely embedded language*[2] for embedding annotation-based language concepts to the host language. As a continuation of our work presented in [32] we now hypothesize that *annotations can be effectively used as an implementation technique for language composition* (pure embedding in particular). We discuss how annotations provide generic framework for pure embedding [10] with options to provide a custom glue code semantics – the author can specify the meaning of annotations' binding, not only the meaning of annotations themselves.

The contributions of this work are as follows[3]:

- observations of corresponding characteristics between source code annotations and formal languages (sections 3, 4, and 5),
- a discussion of discrepancies between annotations and formal languages (and thus an identification of the main specificity of annotations in comparison with formal languages, section 3.2),
- the 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 3.2, elaborated more in the conference version of the paper),
- a definition of an *annotation-based language* (abbreviated: @L) and its aspects from the viewpoint of formal language theory (throughout the whole paper with a summary in section 6),
- an analysis of language composition options using annotations as a generic framework pure embedding (which can be useful for language authors that are considering implementation options for language composition, section 7),
- a discussion of a host language symmetry in annotation-based language pure embedding that happens when both composed languages are annotation-enabled (section 8), and
- recommendations for selecting the right host language in case of a host language symmetry (sections 8.3 and 8.5).

[1] Examples include the FAL language for secure logging [48], a DSL for questionnaires [17] or data visualization [38], a graphical modelling language used for database design [37], or any other domain, e.g., even for describing organizational patterns [14] or malware behavior patterns [47].

[2] Pure embedding as a specific case of language self-extension defined by Erdweg et al. [10].

[3] The first four items are shared with the conference version of the paper, the last three are original contributions of the extended version.

2. Source Code Annotations

An *annotation-enabled language* (abbreviated: @EL) is any formal language that supports attribute-oriented programming (source code annotations). 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 the XDoclet[4] technology). An example of an @EL is the Java programming language from version 1.5. An annotation-enabled language (@EL) is the host language for an annotation-based language (@L), which consists of source code annotations (for the complete definition of @L see section 6)[5].

Custom declarative tags supported by @EL are *source code annotations* (abbreviated: annotations). An annotation annotates an annotation-enabled program element[6]. 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. Our definition of annotations introduced in [32] is presented in Definition 1.

Definition 1 *Annotations are custom declarative structured tags in host @EL that are bound to host program elements using an 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.*

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.

3. 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.

3.1. Structural Correspondence

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 a mapping between Java classes and a relational database. In a simple example presented in Figure 1 an `@Entity` annotation is used to specify

[4] `http://xdoclet.sourceforge.net/xdoclet/index.html`

[5] Therefore, when we will talk about @L, we will be referring to annotations, and when we will talk about @EL, we will be referring to the host language that is annotated with annotations.

[6] An 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).

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[7]. 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.

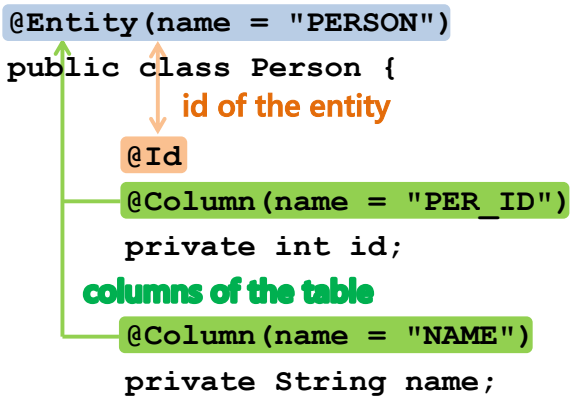

Fig. 1. Mapping of the Person entity class to database using JPA annotations

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.

Now when we look at the given AST, we can easily devise a simple domain-specific language that would to a degree copy the structure of these annotations. Snippet in Listing 1.1 can be an illustration of a DSL that expresses the same information as annotations. We can see that JPA annotations can be considered a configuration DSL, thus we see that there is a correspondence between annotations and formal languages.

Listing 1.1. Person entity definition

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

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

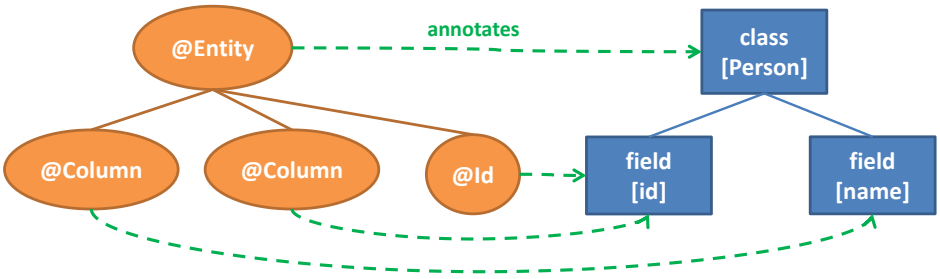

Fig. 2. AST model of the Person entity definition using JPA annotations

In practice we can find multiple structural stereotypes (idioms) in annotations' usage that support the importance of this structural correspondence. Each *idiom specifies a common structural relationship between annotations* (annotation-based language concepts) and as such defines a part of the annotation-based language's grammar. We refer the reader to [32] for the detailed review and discussion of annotations' usage idioms.

3.2. Structural Specificities

We discussed an important observation about the 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, a 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). These restrictions are characteristic for annotations and in language theory they correspond to language composition[8].

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

[8] In this the reader can find a better correspondence with domain-specific aspect languages (DSAL) [12] that are designed to express crosscutting concerns (aspects [46]) and thus are closely coupled with the base language.

– *reversed code-wise relations* specify host language's requirements for the annotations to be used in a context of a given host program element.

Via *code-wise relations*, an annotation may specify requirements for its target program element – the restrictions that have to be kept. Basically, code-wise relations specify on which host program elements annotations can be used.

As a standard example of enforcing existing code-wise relations we can mention Java's `@Target` metaannotation (an 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.

Reversed code-wise relations pose restrictions on the host language according to the annotations that are present in the code. E.g., we can require that the actual method parameter type must be annotated by a particular annotation, e.g., `@Serializable`. In this example we want the method user to use the `@Serializable` annotation to confirm that she is aware that the object sent to our method will be serialized. This requirement should be documented in the API of our method, and our implementation should properly report its violation, so that the user can fix the error. Basically, reversed code-wise relations specify how a host language can 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)[9]. This kind of relations increases integration of annotations into the host language.

As an example we can use the `@Override` Java annotation. 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#).

3.3. Summary

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

[9] All the errors should be well documented in the API of the tools that process annotations, and in case of error, the tool should properly report it.

– @EL concepts' requirements on annotations – **reversed code-wise restrictions**.

If we consider a set of related annotations an *annotation-based language*, then based on the discussion we can define the abstract syntax of an @L as in Definition 2.

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.*

4. Concrete Syntax Correspondence

The annotations author can design their 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 @ELs the apparatus for the concrete syntax definition of annotations are *annotation type*s.

4.1. 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 1.2 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 the second 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", an integer value and a string.

Listing 1.2. C# version of the `Configuration` annotation type

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

Host grammar rules specify CS restrictions on Java annotations. E.g., Java annotations have to start with the '@' sign, followed by the annotation name. Then there are optional annotation parameters enclosed in parentheses, and so on. In a conventional formal language the language author is usually not restricted by any such rules. In this aspect annotations resemble more generic languages[10] [3], such as XML languages, and alike; that are built around a given syntactic skeleton.

Annotation types partially describe annotation-based language model [35] – they define relations between annotations and their parameters (that could possibly be other annotations). However, their current implementations do not allow expressing relations other than nesting of parameters in annotations.

[10] Mernik [27] calls a generic language Commercial Off-The-Shelf (COTS).

4.2. Binding Rules

@L concrete syntax has another specific aspect – *binding rules*. Each annotation *marks* (annotates) its target program element. This relationship is expressed by the relative position of an 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 an unambiguous mapping to its target language element and that for each program element there will be an unambiguous way of finding its annotations. The most common binding for annotations is using annotations as prefixes. E.g., the Java annotations are considered modifiers and therefore they prefix declarations just as Java modifiers do.

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

4.3. Summary

Each particular host @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 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 set of concrete annotation types of annotations that belong to the @L.*

5. Semantics Correspondence

Semantics of the language is the most important part of the language definition – it gives the language a meaning [21]. There are multiple ways of describing the 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* [19]).

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 a pluggable annotation processing API released under JSR 269 specification[11]. From the viewpoint of annotations' semantics this API is usually used to implement M2T transformations [42] – annotation processors usually process the host AST enriched with parsed annotations and they generate new source code as text.

[11] `https://www.jcp.org/en/jsr/detail?id=269`

An alternative to JSR 269 is a Spoon API by Pawlak [33] 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, methods, etc. These can be used to find out whether there is a particular annotation annotating the chosen program element and then act upon it. Although the JSR 269 annotation processor API and reflection API are not the same, in [34] we have shown a unified API bridging those two.

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 a reference implementation using a pluggable annotation processor or GPL code using a reflection API. The reference implementation may use convenience frameworks, such as Google Reflections.*

While this approach to semantics definition seems to be sufficient, we would like to note that there are multiple formal approaches to semantics definition when we are talking about DSLs [22] that tend to be better in some aspects, e.g., in comprehensibility. Therefore, annotation-based languages could benefit from such a formal apparatus. In our future work we want to identify the best @L semantics definition apparatus.

6. Annotation-Based Language

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[12]. For example, if we have a set of JPA annotations used to describe a 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*).*

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:

[12] We should also note that we expect most of the @Ls (if not all) to be domain-specific languages rather than general purpose languages. E.g., annotations dedicated for the feature model driven generation in a particular domain [40].

- @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).

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 the @L sentence to which the annotation belongs.

7. Annotation-Based Language Composition

In general we can consider annotations a *generic purely embedded language*, or a generic framework for pure embedding [10]. It has an analogy in *generic language*s. A generic language provides a syntactic skeleton for a language author where the author can get a generic parser for free. As a drawback she has to accept syntactic restrictions posed upon the language. Analogically, a pure embedding framework provides a syntactic skeleton for embedding that gives us a free parsing of the embedded language. In case of annotations the pure embedding is supported by providing free parsing of the purely embedded @L. @EL has to provide means for processing the annotations in the AST or at least later during the runtime from the program representation using reflection mechanism. On the other hand, the annotation-based pure embedding framework poses syntactic restrictions to @L, both in terms of annotations concrete syntax and binding (pure embedding) restrictions (not everything can be annotated).

Based on the terminology used by Erdweg et al. [10], language composition using annotations is a case of pure embedding, which itself is a case of language self-extension (by the @L definition, its host @EL has to provide a parser that is able to parse also annotations). However, we can distinguish three types[13] of pure embedding using annotations depending on the relationship of composed languages L1 and L2: a) language unification, b) language referencing by extension, and c) language extension. Processing of annotations is implemented as a plug-in to host language parser, or the language is evaluated in runtime using evaluation. The discussion of these options aims to support our hypothesis that *annotations can be used as an implementation technique for language composition*. Therefore, annotations can be considered an implementation and design[14] strategy for the aforementioned language composition types.

We want to note upfront that the presented discussion aims mainly at examining the potential of annotations in the field of language composition. It does not discuss in detail all practical implications of composition implementation using annotations that would be required by language engineers to make a responsible choice when selecting an implementation platform. We consider the comparison study examining these implications as the next step in this research.

7.1. Annotation-Based Language Unification

If the concepts represented by annotations are (or can be) just another representation of a separate standalone non-annotation-based language L2 then we will consider the

[13] Again we try to use terminology from [10] to be as consistent as possible.

[14] Annotations require the embedded language to follow restrictions defined by host @EL, therefore they are not solely the implementation strategy, but also influence @L design.

@L an *annotation-based language unification* of the L1 (L1 is the host language) and L2 (the embedded language). Annotation-based unification is a mapping of L2 concepts to L1 concepts. This mapping defines which L2 concepts annotate which L1 concepts. Annotations provide a glue code skeleton for unification.

Let us consider an explanatory example that we will base on the JPA annotation-based configuration domain-specific language. As the host @EL (L1) for the unification, the JPA uses the Java language. In Java we can define an entity class that represents a Person entity in some web application. The class could look like the code snippet presented in Listing 1.3.

Listing 1.3. Person entity class in Java

```
public class Person {
  private int id;
  private String name;
  private int age;
  ...
}
```

This class is used by the application to represent a Person entity during the runtime in memory. JPA defines a configuration language that is used both for the purposes of object-relational mapping definition, and for database script generation. Originally, the configuration language was based on XML. So in case we wanted to map the Person class to the relational database, we could use the JPA XML configuration in Listing 1.4.

Listing 1.4. XML-based JPA configuration for the Person entity

```
<entity class="model.Person" name="Person">
  <table name="PERSON"/>
  <attributes>
    <id name="id">
      <column name="IDENTIFIER"/>
    </id>
    <basic name="name">
      <column name="NAME"/>
    </basic>
    <basic name="age">
      <column name="AGE"/>
    </basic>
  </attributes>
</entity>
```

Now if we consider this configuration language[15] to be the standalone language L2, we can embed it directly to the Java language (in this case the host L1). We could devise the annotation-based language from scratch, but we will just show the resulting language that is the embedded version of the JPA configuration language[16]. Listing 1.5 shows the annotation-based version of the same JPA configuration using standard JPA annotations.

[15] The whole JPA configuration language is larger then just the sentence presented in Listing 1.4. However, in this paper we will restrict it to the mentioned part.

[16] For a detailed process of designing an annotation-based language from an XML-based one, we kindly refer the reader to the case study presented in [31].

Listing 1.5. Annotation-based JPA configuration

```
package model;

@Entity(name = "Person")
@Table(name = "PERSON")
public class Person {
  @Id
  @Column(name = "IDENTIFIER")
  private int id;

  @Basic
  @Column(name = "NAME")
  private String name;

  @Basic
  @Column(name = "AGE")
  private int age;

  ...
```

The concepts from the XML-based language were transformed into annotations[17] – `@Entity`, `@Table`, `@Column`, etc.

This example illustrates that an annotation-based language can represent a form of an embedded language. The semantics of the embedded JPA @L is not only the database table definition. Also the *binding* between annotations and their target program elements is meaningful; it defines the mapping between classes from object-oriented programming paradigm (host @EL) and tables from relational databases (defined by annotations – @L). The processing of annotations and their binding to host target elements is done by standard tools; in case of JPA implementations it is usually reflection that provides the runtime model of the language sentence (program elements and their annotations).

An example of annotation-based language unification can be found in the work of Cimadamore et al. [5]. They used annotations to embed Prolog theories (L2) within Java classes (L1), and to specify Prolog code as a possible implementation of annotated Java methods.

7.2. Annotation-Based Language Referencing by Extension

Attribute-oriented language referencing by extension is again a composition of two standalone languages L1 and L2. However, annotations in this case do not constitute any of the languages. Let us suppose that we extend L1 with references to L2. Annotations in the L1 would be only references referring to concepts of L2 (referring to an existing sentence of L2). Here again the attribute-oriented referencing defines a mapping of L2 concepts to L1 concepts. However, this time it uses navigational relationship instead of in-place binding. In this situation @OP plays the role of the glue code again.

[17] There is an XML element `attributes` that has no direct counterpart in the annotation-based version of the language. We chose to use the version in Listing 1.5, because it is a valid JPA configuration. Thus in the same time the listing also shows an industrial example of the embedding.

Again, we will illustrate the attribute-oriented referencing with the JPA case study. We will use Java annotations to extend Java (annotation-enabled language L1) with references to the database definition language (referenced language L2). We will consider the `name` XML attributes from the database definition XML language the identifiers of the "entity" and the "column" database concepts. Now instead of introducing all the database definition language concepts as annotations to Java class we will only refer to the database definition concepts. For this purpose we will use the `@EntityRef` and the `@ColumnRef` annotations as show in Listing 1.6.

Listing 1.6. Annotations referencing the database DSL

```
@EntityRef("Person")
public class Person {
  @ColumnRef("IDENTIFIER")
  private int id;

  @ColumnRef("NAME")
  private String name;

  @ColumnRef("AGE")
  private int age;

  ...
```

This @L does not provide all the information from the XML-based configuration example. Rather it just defines the references to the database DSL. These referencing annotations cannot constitute a standalone language – at least not the same as was shown in Listing 1.4. This example misses the information about the database column types – the information that the `id` variable is the primary key of the corresponding table (in Listing 1.4 it was expressed using the `id` and `basic` XML elements). The `@EntityRef` annotation just binds the "Person" entity defined in the XML-based configuration with the `Person` class[18].

An example of annotation-based language referencing can be CoMA annotations [1], @L extending Java programs (L1) to enable references to their formal specification (L2) given in terms of Abstract State Machines.

7.3. Annotation-Based Language Extension

If the @L concepts cannot devise a standalone language *without* the relationships with the concepts of the host language that they annotate, then the @L is an *attribute-oriented language extension* of the host language. Attribute-oriented extension defines a mapping between the concepts of the E language extension to the concepts of the base language L1. The technique of attribute-oriented programming is again a glue code framework, this time for the language extension.

[18] Of course, in this particular example the XML-based configuration already contains links to the Java language, and so the annotation-based referencing is redundant. We used this example only to illustrate the principle of annotation-based referencing.

As a representative example we can mention the `@Override` annotation in the Java language. This annotation is an @L just by itself, it is not dependent on any other annotations. But the concept "Override" cannot be a language itself, it is just a language extension. In general, the "Override" concept expresses a relationship that says that something overrides something else. But without the concepts of the host Java language (target program elements of the `@Override` annotations) the "Override" concept has no meaning. In Listing 1.7 there is an example of using `@Override` extension on the `toString()` method inherited from `java.lang.Object` class.

Listing 1.7. `@Override` @L extension in Java

```
public class Person {
  ...
  @Override
  public String toString() {
    return this.name + " (" + this.age + ")";
  }
}
```

An example of annotation-based language extension is the `@reify` annotation defined by Gerakios et al. [16] that allows annotating generic type arguments to specify by-expansion translation of generics relative to selected type parameters only. Java generics are compiled by erasure: all clients reuse the same bytecode, with uses of the unknown type erased, the `@reify` annotation enables by-expansion translation: each type-instantiation of a template produces a different code definition.

8. Symmetry in Annotation-Based Language Composition

An interesting observation about annotation-based language composition is a symmetry in cases where the annotation-based language composition is used to compose two standalone languages[19] – unification, and referencing by extension. If we consider two languages for composition using annotation-based approach, we have to identify which one of the pair will be the host language for annotations. If only one language of the pair is annotation-enabled then the choice is obvious. On the other hand, if both languages support annotations, then we can make a choice about which language will be the embedded one and which the hosting one.

8.1. Annotations in XML

We can utilize XML Schema for XML language definition to support annotations in a particular XML language, . @OP support can be achieved by adding the `<any>` or `<anyAttribute>` element to all the language concepts that should support annotations. The `<any>` XML schema element enables us to extend the XML document with arbitrary elements that are not specified by the XML schema. The `<anyAttribute>`

[19] Of course, it applies only for cases when both of the standalone languages support attribute-oriented programming – both of them must be annotation-enabled. E.g., since SQL does not support annotations, Java could not be embedded into it using annotations (not to mention that there is probably little reason to embed Java into SQL.

element enables us to extend the XML document with arbitrary XML attributes not specified by the schema. Adding either one of these elements to possible target element types of the annotation-enabled XML language concepts will emulate the attribute-oriented programming for the given XML language.

To illustrate the idea let us have a look at the usage of the `<any>` XML Schema element allowing to add multiple custom declarative tags to the `entity` concept of the JPA XML configuration language. The `entity` element definition with the support for annotations is presented in Listing 1.8.

Listing 1.8. `<any>` element simulating @OP using XML Schema definition

```
<xs:element name="entity">
  <xs:complexType>
    <xs:sequence>
      <xs:any minOccurs="0"/>
      <xs:element name="table" type="tableType"/>
      <xs:element name="attributes" type="attributesType"/>
    </xs:sequence>
    <xs:attribute name="name" type="xsd:string" use="required"/>
    <xs:attribute name="class" type="xsd:string" use="required"/>
  </xs:complexType>
</xs:element>
```

Adding such an extension point to all the annotation-enabled concepts of the JPA XML language enables us to annotate the concepts with custom XML tags. In comparison with Java annotations, our custom tags in XML do not prefix host language concepts, they are rather their children in the XML tree hierarchy (and in this case they are listed first in the sequence). Of course there are multiple other approaches to @OP for XML. We have chosen this particular one as an illustration.

Custom tags added in place of `<any>` element must be XML elements. Thanks to that they can be parsed by the XML parser and can be accessed via XML tools in a standard way, thus meeting our requirements from the annotations definition (definition 1). In the discussion about the symmetry we will employ this approach to embed Java into JPA XML language.

8.2. Symmetry in Annotation-Based Language Unification

In annotation-based language unification a whole sentence of the L2 is embedded into a sentence of the L1 (that is annotation-enabled). If the standalone L2 is annotation-enabled as well, we can switch the roles of L1 and L2 and embed L1 concepts to L2 as annotations.

As an example we will look back at the JPA @L from section 7.1. We have already presented a sentence of this @L and also a corresponding sentence in the standalone XML language. In the case shown in section 7.1 we have seen the database definition language embedded into the Java language. The same way a database definition could be annotated with Java code. The only requirement needed is the support for annotations. In other words, the database definition language has to be annotation-enabled.

Therefore we can define a new @L that will introduce Java concepts to the database definition XML language (using XML Schema) and add the `Person` class specification to its database definition. We will call this language the Java entities @L. The sentence in

this language is listed in Listing 1.9. To distinguish the XML elements of the embedded Java entities @L from the XML elements of the host database definition language, we will use the `java` prefix for `embeddedJava` namespace.

Listing 1.9. JPA XML language annotated with Java concepts

```
<entity name="Person" xmlns:java="embeddedJava">
  <java:class modifier="public" abstract="false">
    Person
  </java:class>
  <table name="PERSON"/>
  <attributes>
    <id>
      <java:field modifier="private" type="int">
        id
      </java:field>
      <column name="IDENTIFIER"/>
    </id>
    <basic>
      <java:field modifier="private" type="java.lang.String">
        name
      </java:field>
      <column name="NAME"/>
    </basic>
    <basic>
      <java:field modifier="private" type="int">
        age
      </java:field>
      <column name="AGE"/>
    </basic>
  </attributes>
</entity>
```

The XML sentence above with the Java entities @L reverses the role of the JPA XML language and the Java language in attribute-oriented language unification. Now the JPA XML language is the host @EL and the entity Java class is the @L. Listing 1.9 expresses the same information as Listing 1.5.

8.3. Host Language Choice in Unification Symmetry

In case of symmetry in language unification, many times the choice of the host @EL is quite easy. If we consider the simple example of the JPA @L and on the other hand the Java entities @L both alternatives might seem quite the same. However, in real world the simple entity Java class only with fields is not a common case. Usually the entity class includes also getters and setters, other utility methods, and even other annotations (e.g., validation annotations). Embedding all these program elements into the XML language would impede its usability (decreasing its readability, etc.). Since a language is also a user interface, its usability[20] is of no less importance [2, 15].

[20] Although sometimes the grammar can be enhanced by refactoring [20], in general refactoring will not remove the consequences of bad choice of host language.

When selecting the right language as the host @EL we should therefore consider the *mutual coverage* of the languages. If the L2 concepts should be embedded to L1 concepts, then all the L2 concepts from the whole sentence have to be mapped to L1. While the JPA XML language concepts can all be mapped to the entity class (as JPA specification proves), not all the entity class concepts can be mapped to the database definition. For example, if we consider the `toString()` method of the entity class, it does not have any counterpart in the database definition. Although we could embed it to the database definition language anyway, the binding between corresponding annotation and its target element would be meaningless. Binding between annotations and target elements should describe significant relationships between @L concepts and @EL concepts. Annotating a database definition concept with the `toString()` method when there is no connection between them indicates a bad smell in annotation-based language design[21].

We should consider other consequences of the choice as well. Here we can refer to the work of Correia et al. [6] who have identified bad smells in annotated code. We can prepare a sample sentence using one language as the host @EL and another sample sentence switching the host @EL. Then we can try to identify the influence of the choice according to bad smells identified by Correira et al. and make a decision according to the obtained results.

8.4. Symmetry in Referencing by Extension

The symmetry in referencing says that if both languages in consideration are annotation enabled, then both of them can have the role of the host language.

In section 7.2 we have designed an @L embedded to Java (L1) referencing to database definition language (L2). If we assume that the JPA XML language is @EL we should be able to design an @L embedded to XML language that will refer to Java entity classes.

We have already shown in section 8.1 how we can implement @OP support for languages based on XML. To make the example a little bit more interesting we could choose a different approach to @OP in XML. As an extension point we can use the `<anyAttribute>` element added to all the extensible concepts. In this case a custom annotation will be represented as the annotated element's XML attribute instead of its child element. Using this approach we restrict the annotation-based languages' authors even more. With the `<any>` element an annotation can have nontrivial structure as an XML element. With the `<anyAttribute>` element an annotation has to be a mere key-value pair. However, for references this is perfectly enough.

Here we could devise a new annotation-based language by specifying our own attributes for the XML language. But we can again refer the reader to the existing XML language for JPA configuration. In the example in Listing 1.4 we have shown a valid sentence in the XML-based JPA configuration. This JPA language uses attributes to refer to Java program elements. The `<entity>` element has the `class` attribute with the entity class reference and the `<id>` and `<basic>` elements have `name` attributes with references to the entity class fields.

[21] Of course, there can be multiple occasions when this relationship can make sense, but not in this particular case. In our example the semantics of the mapping between entity class and database entity is used to provide object-relational mapping. And in this mapping there is no place for the `toString()` method. So the choice should take into consideration the semantics of the mapping between L1 and L2.

8.5. Host Language Choice in Referencing Symmetry

In case of annotation-based referencing the choice of the appropriate host @EL is different than in case of annotation-based unification. Here there is no issue of mutual coverage of the considered languages. Only those bindings have to be expressed that really need to be expressed. If there is a concept from the L2 that does not have a counterpart in the L1, we do not have to add a reference annotation to L1 for that concept. References do not have to cover complete L2 (referenced language).

Still, bad smells identified by Correia et al. [6] can help in decision in case of attribute-oriented referencing too. E.g., if the sentences of the L1 are already crowded with different annotations (Crowded Party bad smell) it might be better to embed reference annotations to L2 instead of L1 in order to prevent crowding the L1 even more.

9. Related Work

The works related to an annotation-based language and components of its definition are discussed in conference version of this paper [32]. In this section we will focus solely on the original content of this paper – topic of language composition.

In section 2 we have stated that a language supporting @OP should support extending its standard grammar with custom declarative marks to annotate standard program elements (in this way it resembles also purely embedded DSLs). From this viewpoint annotation types resemble the island grammars. Island grammars specify only parts (*islands*) of the grammar, not the complete grammar. The rest is called *water* and is left unspecified. They were used for example by Dinkelaker et al. [9] to provide a framework for providing tool-support for embedding DSLs to GPLs, or by Kurš et al. [25] for creating composable and reusable islands called *bounded seas* that compute scope for water parsing. Annotations and island grammars differ in two main aspects. First, annotations are restricted by the host language to their syntactic skeleton. Second, compared to island grammars, annotations are tightly bound to their target program elements. Island grammars do not consider the surrounding source text (water). Therefore the annotations are more restricted in a way they are embedded into the host @EL.

In languages that support higher-order functions a pure DSL embedding is often realized with functions that accept a callable as their argument. This is applicable, for example, for functions accepting code blocks in Ruby and lambdas in Kotlin, or for decorator functions in Python. These functions resemble the role annotations have in the language embedding, as they also represent bindings of the concepts of the embedded language. But unlike annotations, they cannot be used to bind attributes, and what is more, their implementation directly alters program semantics (annotations do not have direct effect on semantics, they have to be processed).

Krahn et al. [24] present MontiCore, a language workbench that supports modularity on the syntax level. In their work they distinguish between two types of language composition: language inheritance, and language embedding. Language inheritance changes parent language's non-terminals, which is obviously not the case of annotations. According to their classification, annotations can be classified as a restricted form of language embedding (restricted on the basis of annotations' binding to their target elements). Another workbench for modular language development is Neverlang [43].

There are multiple language workbenches that support language composition on the semantic level. Workbenches Rascal, Spoofax or SugarJ use composing sets of rewrite rules, Ensõ uses Object grammars based on object-oriented principles [11]. Van der Storm et al. [39] formulated Object grammars as their framework for language extension that supports composition for language modularity. They define three types of composition: language reuse, language extension, and language mixin. Language reuse requires that the embedded language is not modified, which does not hold for composition through annotations (annotations have to follow grammar restrictions posed by @host EL). Annotation-based language extension would be a case of Van der Storm's language extension. Referencing and unification would fall under the language mixin category.

JetBrains MPS[22] is to our best knowledge the most successful language workbench. Voelter [45] discusses MPS' options for language composition and he defines following language modularisation types: extension, combination, reuse, and embedding. From his classification, annotation-based language unification is a case of language embeddings, since @L is syntactically embedded into host @EL. Annotation-based language referencing covers both language combination and language reuse. Voelter's language reuse requires extending the reusable language to allow references to other language. Language combination deals with references between languages, too, however, it requires that the combined languages are designed and implemented with references in mind from the very beginning (thus references are part of the language and not a custom addition as in case of annotations). Annotation-based language extension belongs to language extension category. Voelter discusses MPS implementation of Modular Embedded Language in [44] where he discusses MPS language annotations. MPS language annotations resemble inter-type declarations from aspect-oriented programming. MPS language annotations are language concepts that can extend existing language concepts defined in MPS (thus although implemented as @OP in MPS metalanguage, they do not exploit annotations themselves as a composition framework).

An interesting approach to composition is presented by Diekmann et al. [8] where they define so-called *language boxes*. A language box is something like a slot in a host language, where another language can be embedded (the host language has to define where and which languages can be embedded). The context between boxes is explicitly set by a programmer during typing the source code and is managed by the editor.

We have already mentioned the work of Erdweg et al. [10] that discusses a classification of language composition. Their classification is also used for a case study in [36]. Mernik [27] uses the same classification for a case study of his object-oriented language composition framework in the LISA tool. Furthermore, he provides a comprehensive discussion of language composition terminology.

An alternative to standard annotations are structured comments. Structured comments are for example JavaDoc, xDoclet or XML doc comments on .NET platform. There are many works that use structured comments to emulate annotations, e.g., Flanagan et al. [13] use structured comments to express constraints on the program elements and compliance of code to these constraints is verified by their Extended Static Checker for Java.

Naming and type conventions were extensively used before annotations' introduction to Java [29], e.g. by JUnit framework. The relation between annotations and type and naming conventions is discussed by Tansey et al. [41]. They present a refactoring framework

[22] Meta Programming System, `https://www.jetbrains.com/mps/`

that is able to refactor legacy code using naming and type conventions to use annotations instead (to overcome Version Lock-in and Vendor Lock-in).

An advantage of naming conventions is that they are less verbose. So sometimes their usage may be interesting, but usually they come short in comparison with annotations because of their expressive power [18]. Naming conventions don't allow metadata structuring. And if there are more metadata, naming conventions usage decreases code readability.

10. Conclusion and Future Work

In this paper we have discussed the correspondence we have observed between source code annotations and formal languages. We have reviewed all three aspects of a formal language definition from the viewpoint of annotations, thus providing evidence for the correspondence between annotations and a formal language. We concluded this topic with an *annotation-based language* definition.

In the second half of the paper we continued by discussing annotations from the viewpoint of language composition. We analysed options in using them as a generic pure embedding framework for language composition. The discussion identifies three forms of annotation-based pure embedding – language unification, language referencing by extension, and language extension. These three forms can be implemented using source code annotations as the implementation framework. We have also discussed implementation symmetry in annotation-based language composition in case both composed languages support annotations. We also provided several recommendations how to choose the right host @EL in such a case.

This paper is a basis for further research in the field of annotations from the viewpoint of formal languages and their composition. Our possible future directions include examining our observations in detail using case studies from practice. We want to describe an existing real world annotation-based language from aspects of its concrete syntax, abstract syntax, and semantics. Such a case study should reveal possible shortcomings of annotations from the viewpoint of formal languages. Another direction already mentioned in introduction of section 7 is the comparison study that would analyse practical implications of using annotations as an implementation platform in comparison with other language composition implementation techniques in detail. Comparison of different implementation strategies would be a direct benefit for language engineers considering different implementation strategies for language composition.

Acknowledgments. This work was supported by project KEGA No. 019TUKE-4/2014 Integration of the Basic Theories of Software Engineering into Courses for Informatics Master Study Programmes at Technical Universities – Proposal and Implementation.

References

1. Arcaini, P., Gargantini, A., Riccobene, E.: CoMA: Conformance Monitoring of Java Programs by Abstract State Machines. In: Khurshid, S., Sen, K. (eds.) Runtime Verification, Lecture Notes in Computer Science, vol. 7186, pp. 223–238. Springer Berlin Heidelberg (2012)

2. Bačíková, M., Porubän, J., Lakatoš, D.: Defining Domain Language of Graphical User Interfaces. In: Leal, J.P., Rocha, R., Simões, A. (eds.) 2nd Symposium on Languages, Applications and Technologies. OASIcs, vol. 29, pp. 187–202. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2013)
3. Chodarev, S., Kollár, J.: Extensible host language for domain-specific languages. Computing & Informatics 35(1), 84–110 (2016)
4. Chodarev, S., Lakatoš, D., Porubän, J., Kollár, J.: Abstract syntax driven approach for language composition. Central European Journal of Computer Science 4(3), 107–117 (2014)
5. Cimadamore, M., Viroli, M.: A Prolog-oriented Extension of Java Programming Based on Generics and Annotations. In: Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java. pp. 197–202. PPPJ '07, ACM, New York, NY, USA (2007)
6. Correia, D.A.A., Guerra, E.M., Silveira, F.F., Fernandes, C.T.: Quality Improvement in Annotated Code. CLEI Electron. J. 13(2) (2010), article ID 7
7. Deransart, P., Jourdan, M., Lorho, B.: Attribute Grammars: Definitions, Systems and Bibliography. Springer-Verlag New York, Inc., New York, NY, USA (1988)
8. Diekmann, L., Tratt, L.: Eco: A Language Composition Editor. In: Software Language Engineering, Lecture Notes in Computer Science, vol. 8706, pp. 82–101. Springer International Publishing (2014)
9. Dinkelaker, T., Eichberg, M., Mezini, M.: Incremental concrete syntax for embedded languages with support for separate compilation. Science of Computer Programming 78(6), 615–632 (2013)
10. Erdweg, S., Giarrusso, P.G., Rendel, T.: Language Composition Untangled. In: Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications. pp. 7:1–7:8. LDTA '12, ACM, New York, NY, USA (2012)
11. Erdweg, S., van der Storm, T., Völter, M., Tratt, L., Bosman, R., Cook, W.R., Gerritsen, A., Hulshout, A., Kelly, S., Loh, A., Konat, G., Molina, P.J., Palatnik, M., Pohjonen, R., Schindler, E., Schindler, K., Solmi, R., Vergu, V., Visser, E., van der Vlist, K., Wachsmuth, G., van der Woning, J.: Evaluating and comparing language workbenches. Computer Languages, Systems & Structures 44, Part A, 24–47 (Dec 2015)
12. Fabry, J., Dinkelaker, T., Noyé, J., Tanter, E.: A taxonomy of domain-specific aspect languages. ACM Computing Surveys 47(3), 40:1–40:44 (Feb 2015)
13. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended Static Checking for Java. SIGPLAN Not. 37(5), 234–245 (May 2002)
14. Frt'ala, T., Vranić, V.: Animating organizational patterns. In: Proceedings of the Eighth International Workshop on Cooperative and Human Aspects of Software Engineering. pp. 8–14. CHASE '15, IEEE Press (2015)
15. Galko, L., Porubän, J.: Approaches to human-computer interaction based on observation of a software developer. In: Proceedings of the 2015 IEEE 13th International Scientific Conference on Informatics. pp. 103–108 (Nov 2015)
16. Gerakios, P., Biboudis, A., Smaragdakis, Y.: Reified Type Parameters Using Java Annotations. SIGPLAN Not. 49(3), 61–64 (Oct 2013)
17. Gouseti, M., Peters, C., van der Storm, T.: Extensible language implementation with object algebras (short paper). ACM SIGPLAN Notices 50(3), 25–28 (Mar 2015)
18. Guerra, E., Fernandes, C., Silveira, F.F.: Architectural Patterns for Metadata-based Frameworks Usage. In: Proceedings of the 17th Conference on Pattern Languages of Programs. pp. 1–14. PLoP2010 (2010)
19. Kleppe, A.: A Language Description is More than a Metamodel. In: 4th International Workshop on Language Engineering. ATEM 2007 (2007), `http://doc.utwente.nl/64546/`
20. Kollár, J., Halupka, I., Chodarev, S., Pietriková, E.: pLERO: Language for grammar refactoring patterns. In: 2013 Federated Conference on Computer Science and Information Systems. pp. 1503–1510. FedCSIS 2013 (Sept 2013)

21. Kollár, J., Sičák, M., Spišiak, M.: Towards Machine Mind Evolution. In: 2015 Federated Conference on Computer Science and Information Systems. pp. 985–990. FedCSIS 2015 (Sept 2015)
22. Kosar, T., Bohra, S., Mernik, M.: Domain-Specific Languages: A Systematic Mapping Study. Information and Software Technology 71(C), 77–91 (Mar 2016)
23. Kosar, T., Martínez López, P.E., Barrientos, P.A., Mernik, M.: A Preliminary Study on Various Implementation Approaches of Domain-specific Language. Information and Software Technology 50(5), 390–405 (Apr 2008)
24. Krahn, H., Rumpe, B., Völkel, S.: MontiCore: Modular Development of Textual Domain Specific Languages. In: Objects, Components, Models and Patterns, Lecture Notes in Business Information Processing, vol. 11, pp. 297–315. Springer Berlin Heidelberg (2008)
25. Kurš, J., Lungu, M., Iyadurai, R., Nierstrasz, O.: Bounded seas. Computer Languages, Systems & Structures 44, Part A, 114–140 (2015)
26. Lakatoš, D., Porubän, J., Bačíková, M.: Declarative specification of references in DSLs. In: 2013 Federated Conference on Computer Science and Information Systems. pp. 1527–1534. FedCSIS 2013 (Sept 2013)
27. Mernik, M.: An object-oriented approach to language compositions for software language engineering. Journal of Systems and Software 86(9), 2451–2464 (2013)
28. Mernik, M., Heering, J., Sloane, A.M.: When and How to Develop Domain-specific Languages. ACM Computing Surveys 37(4), 316–344 (Dec 2005)
29. Newkirk, J., Vorontsov, A.: How .NET's custom attributes affect design. Software, IEEE 19(5), 18–20 (Sept 2002)
30. Noguera, C., Pawlak, R.: AVal: an extensible attribute-oriented programming validator for Java: Research Articles. Journal of Software Maintenance and Evolution 19(4), 253–275 (Jul 2007)
31. Nosáľ, M., Porubän, J.: XML to Annotations Mapping Definition with Patterns. Computer Science and Information Systems 11(4), 1455–1477 (2014)
32. Nosáľ, M., Sulír, M., Juhár, J.: Source Code Annotations as Formal Languages. In: 2015 Federated Conference on Computer Science and Information Systems. pp. 953–964. FedCSIS 2015 (Sept 2015)
33. Pawlak, R.: Spoon: Compile-time Annotation Processing for Middleware. IEEE Distributed Systems Online 7(11), 1–13 (Nov 2006)
34. Pigula, P., Nosáľ, M.: Unified Compile-Time and Runtime Java Annotation Processing. In: 2015 Federated Conference on Computer Science and Information Systems. pp. 965–975. FedCSIS 2015 (Sept 2015)
35. Porubän, J., Chodarev, S.: Model-aware language specification with Java. In: 2015 13th International Conference on Engineering of Modern Electric Systems. pp. 1–4. EMES 2015, IEEE (2015)
36. Reis, L.V., Di Iorio, V.O., Bigonha, R.S.: An on-the-fly grammar modification mechanism for composing and defining extensible languages. Computer Languages, Systems & Structures 42(C), 46–59 (Jul 2015)
37. Ristić, S., Aleksić, S., Čeliković, M., Luković, I.: Generic and Standard Database Constraint Meta-Models. Computer Science and Information Systems 11(2), 679–696 (2014)
38. Smeltzer, K., Erwig, M., Metoyer, R.: A transformational approach to data visualization. ACM SIGPLAN Notices 50(3), 53–62 (Mar 2015)
39. van der Storm, T., Cook, W.R., Loh, A.: The design and implementation of Object Grammars. Science of Computer Programming 96(4), 460–487 (2014)
40. Táborský, R., Vranić, V.: Feature Model Driven Generation of Software Artifacts. In: 2015 Federated Conference on Computer Science and Information Systems. pp. 1007–1018. FedCSIS 2015 (Sept 2015)
41. Tansey, W., Tilevich, E.: Annotation refactoring: inferring upgrade transformations for legacy applications. SIGPLAN Not. 43(10), 295–312 (Oct 2008)

42. Đukić, V., Luković, I., Popović, A., Ivančević, V.: Model Execution: An Approach based on extending Domain-Specific Modeling with Action Reports. Computer Science and Information Systems 10(4), 1585–1620 (2013)
43. Vacchi, E., Cazzola, W.: Neverlang: A framework for feature-oriented language development. Computer Languages, Systems & Structures 43, Part C, 1–40 (2015)
44. Voelter, M.: Implementing Feature Variability for Models and Code with Projectional Language Workbenches. In: Proceedings of the 2nd International Workshop on Feature-Oriented Software Development. pp. 41–48. FOSD '10, ACM, New York, NY, USA (2010)
45. Voelter, M.: Language and IDE Modularization and Composition with MPS. In: Generative and Transformational Techniques in Software Engineering IV, Lecture Notes in Computer Science, vol. 7680, pp. 383–430. Springer Berlin Heidelberg (2013)
46. Vranić, V., Kuliha, B.: Realizing changes by aspects at the design level. In: Proceedings of the 2015 IEEE 19th International Conference on Intelligent Engineering Systems. pp. 369–374. INES 2015 (Sept 2015)
47. Šťastná, J., Tomášek, M.: Exploring malware behaviour for improvement of malware signatures. In: Proceedings of the 2015 IEEE 13th International Scientific Conference on Informatics. pp. 275–280 (Nov 2015)
48. Zawoad, S., Mernik, M., Hasan, R.: Towards Building a Forensics Aware Language for Secure Logging. Computer Science and Information Systems 11(4), 1291–1314 (2014)

Milan Nosáľ is an Assistant Professor at the Department of Computers and Informatics, Technical University of Košice, Slovakia. He received his PhD. in Computer Science in 2015 for his work 'Leveraging Program Comprehension with Concern-oriented Projections'. Currently his research focuses on attribute-oriented programming (source code annotations), program comprehension, projectional editors, and domain-specific languages.

Matúš Sulír is a PhD student at the Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice. He graduated with a master's degree in Computer Science in 2014. His current research is focused on program comprehension, source code annotations, and empirical methods in software engineering.

Ján Juhár is a PhD student at the Department of Computers and Informatics, Technical University of Košice, Slovakia. He received his MSc in Computer Science in 2014. His current research focuses on program comprehension, programming tools, and projectional editors.

Received: January 14, 2016; Accepted: June 1, 2016.