JUMP—From Java Annotations to UML Profiles*

Alexander Bergmayr[1], Michael Grossniklaus[2], Manuel Wimmer[1], and Gerti Kappel[1]

[1] Vienna University of Technology, Austria
[lastname]@big.tuwien.ac.at
[2] University of Konstanz, Germany
michael.grossniklaus@uni-konstanz.de

Abstract. The capability of UML profiles to serve as annotation mechanism has been recognized in both industry and research. Today's modeling tools offer profiles specific to platforms, such as Java, as they facilitate model-based engineering approaches. However, the set of available profiles is considerably smaller compared to the number of existing Java libraries using annotations. This is because an effective mapping between Java and UML to generate profiles from annotation-based libraries is missing. In this paper, we present *JUMP* to overcome this limitation, thereby continuing existing mapping efforts by emphasizing on annotations and profiles. We demonstrate the practical value of *JUMP* by contributing profiles that facilitate reverse-engineering and forward-engineering scenarios for the Java platform. The evaluation of *JUMP* shows that profiles can be automatically generated from Java libraries exhibiting equal or even improved quality compared to profiles currently used in practice.

Keywords: Java Annotations·UML Profiles·Model-Based Engineering·Forward Engineering·Reverse Engineering

1 Introduction

Since the introduction of the UML profile mechanism, numerous profiles have been developed [38], many of which are available by the OMG standardization body [36]. Even in industry, the practical value of profiles has been recognized as today's modeling tools offer already predefined stereotypes covered by such profiles. They are considered as a major ingredient for current model-based software engineering approaches [6] by providing features supplementary to the UML standard metamodel. This powerful capability of profiles can also be exploited in terms of an annotation mechanism [42], where defined stereotypes show similar capabilities as annotations in Java. Hence, deriving stereotypes from established programming libraries to produce corresponding profiles at the modeling level is desirable. For instance, IBM's Rational Software Architect provides profiles for certain Java libraries. By applying such profiles, high-level platform-independent models (PIMs) are refined into models specific to a platform (PSMs), where the platform refers to the library from which the profile was derived. Turning this forward-engineering (FE) perspective into a reverse-engineering (RE) one, existing

* This work is co-funded by the European Commission under the ICT Policy Support Programme, grant no. 317859.

programs can be represented as UML models that capture annotations by applying the corresponding profiles. Therefore, platform-specific profiles and their application are beneficial from both perspectives. In a reverse-engineering step, model analyzers can exploit captured stereotypes to facilitate comprehension [10], whereas profiled UML models, i.e., models to which profiles are applied, pave the way for model transformers to generate richer program code in a forward-engineering step [42].

Problem. However, to date, an effective conceptual mapping between UML and Java as a basis for an automated process to generate profiles from libraries that use annotations is still missing. As a result, profiles need to be manually developed, which is only achievable by a huge effort when considering the large number of possible annotations in Java. In the ARTIST project [4], we are confronted with this problem, as we work towards a model-based engineering approach for modernizing applications by novel cloud offerings, which involves representing PSMs that refer to the platform of existing applications, e.g., the Java Persistence API (JPA), when considering persistence, and the platform of "cloudified" applications, e.g., the Objectify library[3], when considering cloud datastores. For instance, JPA annotations at the modeling level facilitate distinguishing between plain association and composition relationships and precisely deciding on multiplicities, which is in general not easily to grasp [7]. UML models profiled by Objectify annotations enable generating method bodies even from a structural viewpoint. These examples highlight the practical value of platform-specific reverse-engineering and forward-engineering tools, which are developed in the ARTIST project.

Contribution. In this paper, we present a fully automatic transformation chain for generating UML profiles from Java libraries that use annotations. For that reason, we propose an effective conceptual mapping between the two technical spaces [25, 30]. Thereby, we continue the long tradition of investigating mappings between Java and UML [15,23,28,33]. Though, in this work, we also consider Java annotations and UML profiles in the mapping process. This necessitates overcoming existing heterogeneities that, e.g., refer to the target specification of Java annotations and other peculiarities of how Java annotation types are declared. To operationalize the conceptual mapping, we employ model transformation techniques [12] as a basis for our approach *JUMP*, which allows developers to "jump" from annotation-based Java libraries to UML profiles. We collect all the automatically generated profiles and make them publicly available in terms what we call the *UML-Profile-Store* [43], thereby complementing OMG's collection of standardized profiles with supplementary profiles for the Java platform.

Structure. In Section 2, we motivate the practical value of platform-specific profiles by a typical *JUMP* use-case and we give the background for *UML Profiles* and *Java Annotations* in terms of metamodels. We present *JUMP* in Section 3 by providing insights into our proposed conceptual mapping and elaborating effective solutions to overcome existing heterogeneities of the two languages. In Section 4, we discuss our prototypical implementation based on the Eclipse ecosystem, while in Section 5, we evaluate *JUMP*. In particular, we (i) compare our methodology how to represent annotations and annotation types in UML with methodologies used in current modeling tools and (ii) evaluate the quality of automatically generated profiles compared to profiles used in practice. Finally, in Section 6, we discuss related work and conclude in Section 7.

[3] https://code.google.com/p/objectify-appengine

2 Motivation and Background

To motivate the practical value of platform-specific profiles, we introduce a typical *JUMP* use-case. Then, we discuss the concepts of Java's annotation mechanism and briefly introduce UML's profile mechanism to establish the basis for our approach.

2.1 Application of Platform-Specific UML Profiles

A typical *JUMP* use-case is directed to scenarios in the context of reverse-engineering (RE) and forward-engineering (FE). They are of particular relevance for migration projects, which aim at reinterpreting existing reengineering processes [26] in the light of advanced model-based engineering approaches [17]. In this respect, UML profiles play an important role as they enable models annotated with platform-specific information [39]. To demonstrate a concrete use-case, we selected the JPA and Objectify profile from the area of data modeling. The idea is to replace the former profile by the latter one, thereby realizing a change of the data access platform as typically required by "moving-to-the-cloud" scenarios. Figure 1 depicts an excerpt of the PSMs of a typical eCommerce web application, where the platform refers to the selected profiles. From the JPA-based PSM, a sliced PIM is generated that sets the focus solely on the domain classes, i.e., annotated with JPA stereotypes, which are intended to be modified. Even better, this generated PIM interprets JPA stereotypes in terms of native UML concepts. As a result, the accuracy of the PIM is improved because it explicitly captures *identifiers*, *compositions*, and more precise *multiplicities*. These improvements of the PIM demonstrate the practical value of considering platform-specific information in the context of a model-based RE scenario. Furthermore, they leverage the refinement

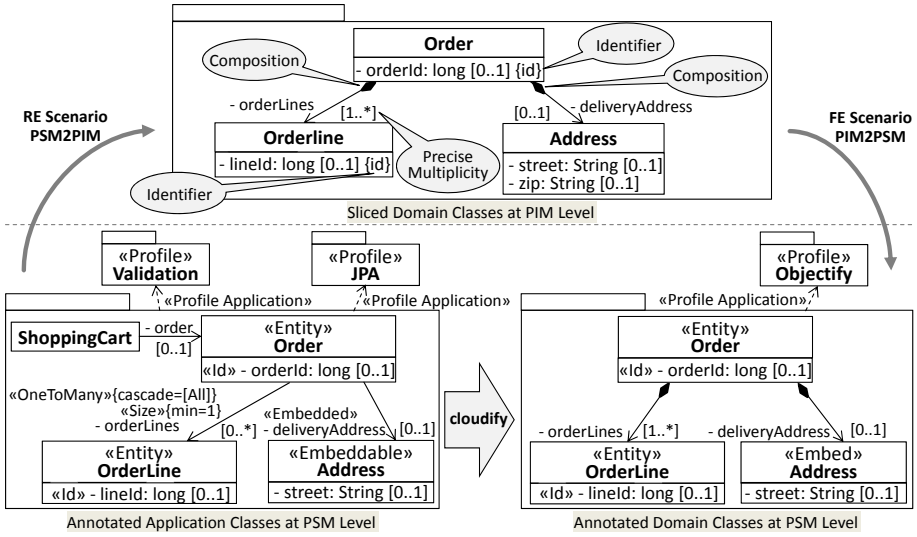

Fig. 1: Typical *JUMP* Use-Case

of the PIM towards an Objectify-based PSM without the need to identify mappings between the pertinent platforms. From the produced Objectify-based PSM, program code can be generated by also interpreting applied stereotypes in the context of a FE scenario. For instance, method bodies for CRUD operations can be generated for domain classes as they are indicated by the respective stereotypes and generated code elements can be automatically annotated. Clearly, *JUMP* acts as an enabler for both RE and FE scenarios by providing the required platform-specific profiles.

2.2 Mechanisms for Annotations in Java and Profiles in UML

Before annotations can be applied on code elements, they need to be declared in terms of annotation types. A rough overview of the main concepts behind annotations in Java is given in the metamodel depicted in Figure 2a. We extracted this metamodel from the JLS7 [37]. `AnnotationTypes` declare the possible annotations for code elements and may have, similar to Java interface declarations, optional `modifiers`. They are identified by a `name`. `AnnotationTypes` may themselves be subject for annotations. Most importantly for the context of this work is the target annotation that is represented in the metamodel as an attribute for simplicity reasons. It indicates the code elements that are valid bases for an application of an `AnnotationType`. The body of an annotation type declaration consists of zero or more `AnnotationTypeElements` for holding information of `AnnotationType` applications. They are declared in terms of method signatures with optional `modifiers`, a mandatory `type` and `name`, and an optional `default` value that is returned if no custom value is set.

With the introduction of UML 2, the profile mechanism has been significantly improved compared to the beginnings of UML [18]. In particular, a profile modeling language has been incorporated in the UML language family to precisely define how profiles are applied on UML models. Figure 2b depicts the core elements of UML's `Profiles` package and relates them to the `Classes` package of UML. As the `Stereotype` metaclass specializes the `Class` metaclass, it inherits modeling capabil-

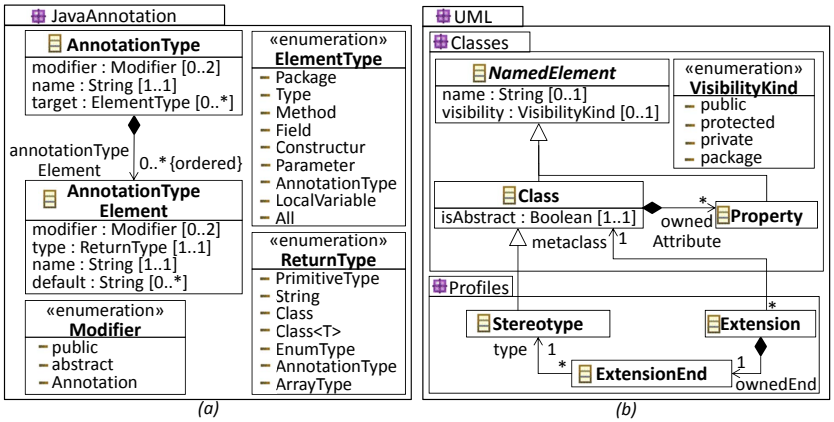

Fig. 2: Metamodel of Java Annotations and UML Profiles

ities such as properties. Defined stereotypes reference the metaclasses that are extended by the `Extension` relationships. The `ExtensionEnd` realizes the reference from the extended `metaclass` back to the `Stereotype`. Similar to `AnnotationTypes`, `Stereotypes` are identified by a `name` property, and modified by an optional `visibility` and the mandatory `isAbstract` property.

To demonstrate the relationship between annotations and stereotypes, we set the focus on the `Order` class of the JPA-based PSM in Figure 1. Listing 1.1 shows the *application* of the `Entity` annotation type to the `Order` class whereas Listing 1.2 depicts the respective *declaration* at the programming level.

Listing 1.1: Application of `Entity`

```
package ...;
import javax.persistence.Entity;

@Entity(name = "Order")
public class Order {
  ...
}
```

Listing 1.2: Declaration of `Entity`

```
package javax.persistence;
import java.lang.annotation.*;

@Target(ElementType.TYPE)
public @interface Entity {
  String name() default "";
}
```

The corresponding UML-based representation is presented in Figure 3, which demonstrates the stereotype application to the `Order` class and the `Entity` declaration by a `Stereotype`. Similarly, at the package-level, the UML profile, which covers the `Entity` stereotype needs to be applied to the `Order`'s package as a prerequisite for the stereotype application. To ensure that the `Entity` stereotype provides at least similar capabilities as the corresponding annotation type, the extension relationship references the UML metaclass `Type`. Furthermore, the stereotype comprises a property corresponding to the annotation type element `name` of the `Entity`.

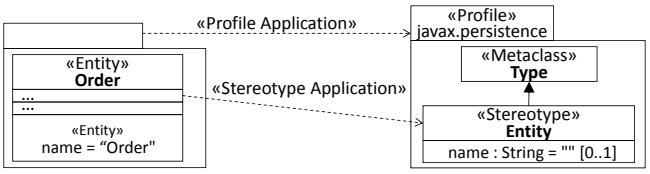

Fig. 3: Application and Definition of `Entity` Stereotype

3 UML Profile Generation from Annotation-based Java Libraries

We start our investigation for generating UML profiles from annotation-based Java libraries by presenting the process of *JUMP*, as shown in Figure 4. The entry-point to *JUMP* is *Java Code* that is translated into a corresponding *Code Model*, which is considered as a one-to-one representation of *Java Code*, i.e., the transition from a text-based to a model-based representation expressed in terms of MOF [35]/EMF [14]. The *Code Model* is the basis for generating a *UML Profile*, which facilitates to capture Java annotation type declarations in terms of UML stereotypes (cf. middle of Figure 4). In turn,

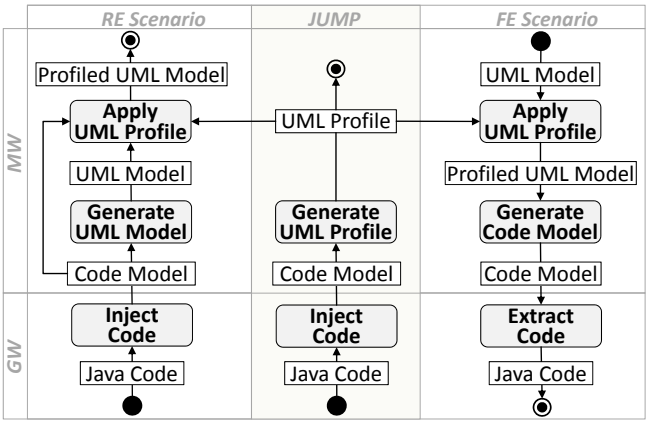

Fig. 4: Process for UML Profile Generation and their Application

they serve as foundation to apply profiles as an annotation mechanism [42]. In case of reverse-engineering *Java Code* (cf. left hand side of Figure 4), the *Profiled UML Model* results from applying profiles to the generated *UML Model*, where the *Code Model* covers the annotated elements that indicate to which elements of the *UML Model* the corresponding stereotypes are applied. Similarly, in case of forward-engineering *Java Code* (cf. right hand side of Figure 4), profiles are applied to the *UML Model* even though, in this case, the *Profiled UML Model* serves as input for generating the *Code Model* from which *Java Code* is extracted. Bridging the two technical spaces [25] we are confronted with, i.e., GrammarWare (GW) [27] and ModelWare (MW) [30], is required for the two scenarios as well as *JUMP*.

3.1 Bridging Technical Spaces

Transforming plain Java code into a UML-based representation requires overcoming the different encoding and resolving language heterogeneities. Concerning the first aspect, the Java code needs to be encoded according to the format imposed by the modeling environment [5]. Concerning the second aspect, a bridge between Java and UML based on translations requires a conceptual mapping between the two languages. Instead of directly translating plain Java code into a UML-based representation, the use of a two-step approach is preferable [24], which is also applied by *JUMP*. In a first step, *Java Code* is translated into a *Code Model* that uses Java terminology and structures conforming to the Java metamodel provided by MoDisco [9]. This *Code Model* is the basis for generating UML profiles and input for the second step that is dedicated to resolving language heterogeneities by relying on the correspondences between the Java and UML metamodels.

3.2 Generating UML Profiles

To facilitate the generation of UML profiles, we present a conceptual mapping between Java's annotation concept and the concept of profiles in UML. Thereby, stereotypes play

a vital role for representing annotation types at the modeling level as they enable their application in a controlled UML standard-compliant way. From a language engineering perspective, stereotypes only extend the required UML metaclasses and facilitate defining constraints and model operations, such as model analysis or transformations, because they can directly be used in terms of explicit types similar to a metaclass in UML. Our proposed mapping is generic in the sense that any declared annotation type can be represented by a stereotype.

Java Concept	UML Concept
AnnotationType a	**add** Stereotype s
a.name	s.name **=** a.name
a.annotationTypeElement	**add** Property p **for each** AnnotationTypeElement **in** a.annotationTypeElement
switch(a.modifier)	
case : public	s.visibility **=** public
case : abstract	s.isAbstract **=** false
case : annotation an **and** **not** an.type = Target	**apply** Stereotype **for** an.type **to** s
case : annotation an **and** an.type = Target	**add** Property p **for each** ElementType **in** a.target p.name = "base_".concat(p.type) **add** Extension e **for each** ElementType **in** a.target e.metaClass = p.type **add** ExtensionEnd f f.type = s
switch(a.target)	
case : AnnotationType	p.type = Stereotype
case : Constructor	p.type = Operation **add** Constraint {self.base_Operation.oclIsDefined() **implies** self.base_Operation.name = self.base_Operation.oclContainer().oclAsType(uml::Classifier).name}
case : Field	p.type = {EnumerationLiteral, Property}
case : LocaleVariable	p.type = Property
case : Method	p.type = {Operation, Property} **add** Constraint {self.base_Property.oclIsDefined() **implies** self.base_Property.oclContainer().oclIsTypeOf(uml::Stereotype)}
case : Package	p.type = Package
case : Parameter	p.type = Parameter
case : Type	p.type = Type **add** Constraint {self.base_Type.oclIsDefined() **implies** Set{uml::Stereotype,uml::Class,uml::Enumeration,uml::Interface} ->>br>includes(self.base_Type.oclType())}
case : none	-- *no Property p needed*
case : all	p.type = {Class, Enumeration, Interface, Operation, Package, Parameter, Property, Stereotype}
AnnotationElementType a	**add** Property p
a.name	p.name = a.name
a.default	p.default = a.default
switch(a.modifier)	
case : public	p.visibility = public
case : abstract	-- *no corresponding feature*
case : annotation an	**apply** Stereotype **for** an.type **to** p
switch(a.type)	
case : PrimitiveType	p.type = uml::PrimitiveType **for** a.type
case : Class	p.type = uml::Class
case : Class<T>	p.type = uml::Class **apply** javaProfile::JGenericType Stereotype **to** p
case : EnumType	p.type = uml::Enumeration
case : AnnotationType	p.type = uml::Stereotype
case : ArrayType	-- *infer lower and upper bound multiplicities*

Table 1: Mappings between Java Annotations and UML Profiles

***AnnotationType* → *Stereotype*.** The mapping presented in the upper part of Table 1 serves as a basis to generate a `Stereotype` from an `AnnotationType`. Thereby, not only its signature needs to be considered but also Java's `Target` meta-annotation. It determines the set of code elements an annotation type is applicable to. The `name` and, with two exceptions, the defined `modifiers` of an `AnnotationType` can straightforwardly be mapped to UML. First, the `abstract` modifier would lead to `Stereotypes` that cannot be instantiated if directly mapped. The problem is caused by Java's language definition. Although the `abstract` modifier is supported to facilitate one common type declaration production rule, it does not restrict the application of `AnnotationTypes`. To ensure the same behavior on the UML level, we never declare a `Stereotype` to be abstract. Second, because annotations are considered as modifiers, it needs to be ensured that the `Target` annotation is properly treated. In fact, the defined set of Java `ElementTypes` determines the required set of `Extensions` to UML meta-classes that specify the application context of the stereotypes.

Generally, most Java `ElementTypes` correspond well to one or more UML meta-classes. Still, constraints are required for some `ElementTypes` to precisely restrict the application scope of the generated `Stereotype` according to their intention. UML does not explicitly support a constructor meta-class. The workaround is to map the `Constructor` to `Operation` and introduce a constraint that emulates the naming convention for constructors in Java. Note that annotation types can have several target types. Thus, before validating the OCL constraint, we have to check which target is actually used in the application. Similarly, the mapping of Java methods to UML requires a constraint as a declared method of an `AnnotationType`, i.e., `AnnotationTypeElement`, is mapped to a `Property` rather than an `Operation` in UML. This is because such methods do not provide a custom realization but merely return their assigned value when they get called. `Properties` in UML provide exactly this behavior. Hence, the constraint ensures that stereotypes generated from annotation types that target Java methods are applicable also to `Property` if they are contained by a Stereotype. Finally, we use a constraint to overcome the heterogeneity of Java's and UML's scope of `Type`. Consequently, stereotypes that extend `Type` are constrained to those elements that correspond to the set of elements generalized by Java's `Type`: `AnnotationType`, `Class`, `Enumeration` and `Interface`. The clear benefit of this approach is a smaller number of generated extension relationships between stereotypes and meta-classes in the profile.

***AnnotationTypeElement* → *Property*.** `AnnotationTypeElements` are mapped to `Properties` as depicted in the lower part of Table 1. Except for the fact that UML properties cannot be defined as abstract, `AnnotationTypeElements` straightforwardly correspond to `Properties`. As `AnnotationTypes` in Java cannot explicitly inherit from super-annotations, the abstract modifier is rarely used in practice. To fully support all return types of `AnnotationTypeElements`, we introduce a `Stereotype` to properly address the fact that `java.lang.Class` provides generic capabilities, which is not the case for UML's meta-class `Class`. Hence, we apply our custom `JGenericType` stereotype to properties with return type `Class<T>`.

4 Implementation and Collected Profiles

To show the feasibility of *JUMP*, we implemented a prototype based on the Eclipse ecosystem. We developed three transformation chains— *JavaCode2UMLProfile*, *JavaCode2ProfiledUML*, and *ProfiledUML2JavaCode*—to realize *JUMP* and the RE and FE scenarios introduced in Figure 1. For injecting *Java Code*, we employed MoDisco [9]. Hence, *JUMP* can be considered as a model discoverer to extract UML profiles from Java libraries. To realize the FE scenario, we extended the Java-based transformer provided by Obeo Network[4]. The prototype and the collection of profiles that we have generated for the evaluation of *JUMP* is available at the *UML-Profile-Store* [43]. It covers 20 profiles, comprising in total over 700 stereotypes. To share these profiles with existing community portals, we submitted them also to ReMoDD [16].

5 Evaluation

The evaluation of *JUMP* is twofold. First, we compare it with existing modeling tools regarding their representational capabilities for dealing with the declaration and application of Java annotation types. Second, we compare UML profiles automatically generated by *JUMP* with UML profiles delivered by IBM's Rational Software Architect. Thereby, our focus is on estimating the quality of the generated UML profiles.

5.1 Methodological Evaluation

As several commercial and open-source modeling tools provide modeling capabilities for UML and the Java platform, the aim of this study is to investigate on their methods for dealing with the application and declaration of annotations. For that reason, we set the focus on a Java-based reverse-engineering example that includes annotations and their declarations. We aim to answer the following research question $(RQ1)$.

RQ1*: What are the methods of current modeling tools to represent Java annotation types and their applications in UML and what are the practical implications?*

To answer $RQ1$, we define a set of comparison criteria that mainly address (i) how the conceptual mapping between Java and UML for annotations is achieved by current modeling tools and (ii) the generative capabilities of these tools regarding profiles. Based on the defined criteria, we evaluate six representative modeling tools and JUMP.

Comparison Criteria. As there are different approaches on how annotation types and their applications are represented at the modeling level, the first and the second comparison criteria ($CC1$ and $CC2$) refer exactly to these extensional capabilities. The third criterion ($CC3$) refers to the support of generative capabilities regarding profiles.

$-CC1$: How are Java annotations applied to UML models?
$-CC2$: How are Java annotation type declarations represented in UML?
$-CC3$: Is the generation of UML profiles from Java code supported?

Selected Tools. We selected six major industrial modeling tools that claim to support reverse engineering capabilities for Java and UML, as summarized in Table 2.

Evaluation Procedure. We defined a simple reference application [43] that declares

[4] http://marketplace.eclipse.org/content/uml-java-generator

a Java class to which we applied an annotation type from an external library. For the purpose of importing the application, we activated the offered functionality of the modeling tools required for a reverse-engineering scenario from Java to UML. While some of the modeling tools are delivered with standard configurations, other modeling tools allow configurations to change the reverse-engineering capabilities by using specific wizards. Moreover, some modeling tools go one step further and allow modifications on the transformation scripts used for the import of Java code. We evaluated the capabilities of the modeling tools offered in the standard settings and explored the different wizard configurations if supported, but we restrained from modifying transformation scripts.

Results. The results of our comparison are summarized in Table 2. Regarding the mapping between Java annotations and UML, we identified that the investigated modeling tools apply one of three significantly different approaches: (i) annotations are considered as a *built-in* feature of the modeling tool, (ii) a *generic* profile for Java is provided, which enables capturing annotations and their type declarations, and (iii) profiles are offered, which are *specific* to a Java library or even an application with custom annotation type declarations. Modeling tools with built-in support for annotations allow their application to arbitrary elements and so to UML elements. Clearly, such an approach facilitates to capture Java annotations, though the type declaration of the annotation in terms of a UML element and its application are not connected. The genericity of this approach, which goes beyond UML models, is clearly one reason for such a behavior. Providing a generic profile for Java means that the modeling tool emulates the representational capabilities of Java, which includes annotations. Although with this approach, the connection of annotation type declarations and their applications can be ensured, the native support of UML for annotating elements with stereotypes is still neglected. However, explicitly defined stereotypes for declared annotation types facilitate their reuse in a UML standard-compliant way and allow model operations to directly exploit them. With specific profiles for Java annotation types, these drawbacks can be overcome. While all evaluated modeling tools provide support for generating profiled UML class diagrams, none of them is capable of generating profiles from Java code.

Modeling Tool			Mapping (Java -> UML)		UML Profile Generation
Name	Version	Availability	Annotation Application	Annotation Declaration	
Visual Paradigm www.visual-paradigm.com	10.2	commerical free community edition	Built-in Tool Feature	Class	-
Rational Software Architect www.ibm.com/developerworks/rational/products/rsa	8.5.1	commerical free for academice use	Specific Profiles	Stereotype	-
Magic Draw www.nomagic.com	17.0.4	commerical free trial version	Generic Java Profile	Interface	-
Enterprise Architect www.sparxsystems.com	9.3	commerical free for academice use	Built-in Tool Feature	Interface	-
Altova UML www.altova.com/umodel.html	2013	commerical free for academice use	Generic Java Profile	Interface	-
ArgoUML argouml.tigris.org	0.34	open-source	Generic Java Profile	Interface	-
JUMP	1.0.0	open-source	Specific Profiles	Stereotype	+

Table 2: Comparison Results

5.2 Quality Evaluation

As UML profiles are already offered by current modeling tools, the aim of this study is to investigate their quality in comparison with profiles automatically generated by *JUMP*. For that reason, we conducted a positivist case study [32] based on real-world Java libraries to evaluate the commonalities and differences between generated profiles and profiles used in practice by following the guidelines of Roneson and Hörst [41]. In this study, we aim to answer the following research question $(RQ2)$.

RQ2*: How is the quality of UML profiles automatically generated from annotation-based Java libraries compared to UML profiles used in practice?*

To answer $RQ2$, we define the requirements of the case study, briefly mention the used Java libraries, and specify the measures based on which the comparison is conducted. Then, we discuss the results of our study not only from a syntactic perspective, but also from a semantic one. The rationale behind this two-step approach is that even though a syntactical matching process for comparing the profiles provides already valuable results, some interesting correspondences may still be uncovered because of potential syntactical and structural heterogeneities [46] between the compared profiles and the conservative matching strategy applied for the syntactical comparison.

Case-Study Design. To conduct this study, the source code of Java libraries that exploit annotations is required. Furthermore, we require existing profiles that claim to support the selected Java libraries at the modeling level. To accomplish an appropriate coverage of different scenarios, the selected Java libraries ideally comprise different intrinsic properties with respect to the design complexity and exploited language elements. Unfortunately, profiles specific to Java libraries in reasonable quality are rarely available. Consequently, in the process of selecting the Java libraries for this study, we were also confronted with the actual offering of modeling tools. IBM's Rational Software Architect (RSA) is obviously close to *JUMP* and offers several profiles of well-known Java libraries mainly for code generation purposes. Thus, we conducted this study by relying on profiles of RSA in version 8.5.1. We selected four established Java libraries for which the source code is available and a corresponding RSA profile in the same major version is offered: Java Persistence API (JPA), Enterprise Java Beans (EJB), Struts and Hibernate. RSA offers them in a UML standard-compliant way. Consequently, we could directly compare them without an intermediate conversion step. All the case-study data including the Java libraries and the profiles are available at our project web site [43].

Case-Study Measures. The measures used in the case study are based on model comparison techniques [29]. Thus, we are interested in equivalent elements that reside in our generated profiles and in the RSA profiles, elements that reside in both solutions but still show differences in their features, and elements that are only available in one of the compared solutions. The measures for estimating the quality of the generated profiles are collected in a two-step matching process. While the first step automatically collects measures based on syntactic model comparison, the second step relies on manually processing differences produced in the first step to deal with semantic aspects.

In the syntactic model comparison, we compute the following measures for certain model elements. To determine element correspondences, we employ as matching heuristic name equivalence, i.e., only if two elements have completely the same

name, they are considered to be corresponding. If an element has no name, such as the `Extension` relationship, it is considered that the elements are corresponding if their source and target elements correspond. Finally, fine grained comparison of the feature values for the given elements is performed. Regarding model elements, we set the focus on (i) `Stereotypes` that are common to both and unique either to *JUMP* or RSA, (ii) differences regarding the `Extensions` of common `Stereotypes`, and (iii) differences regarding the `Properties` such `Stereotypes` cover.

In the semantic model comparison, we take the syntactical differences as input and aim at finding additional correspondences between elements which are hardly explored by a pure syntactic comparison due to the conservative matching strategy. We investigate unmatched elements, especially stereotypes, in our generated profiles and in the RSA profiles, and reason about possible element correspondences beyond String equivalences. Finally, in the semantic processing, we further evaluate the correspondences found in the first phase due to the potential syntactical and structural heterogeneities.

Results. We now present the results of applying *JUMP* to the four selected Java libraries and compare them to the profiles offered by RSA. The full results are also available at our project web site [43]. The absolute number of generated stereotypes by *JUMP* and the provided ones by RSA are depicted in Figure 5a. Figure 5b summarizes (i) the number of stereotypes generated by *JUMP* but not covered by the RSA profiles, (ii) the number of stereotypes that are exclusively covered by the RSA profiles, and (iii) the number of stereotypes that are common to both. These results include correspondences between stereotypes detected throughout the syntactic and semantic comparison. For instance, the EJB profile of RSA covers stereotypes that refer to the `@Local` and `@Remote` annotations of the EJB library, though their signature

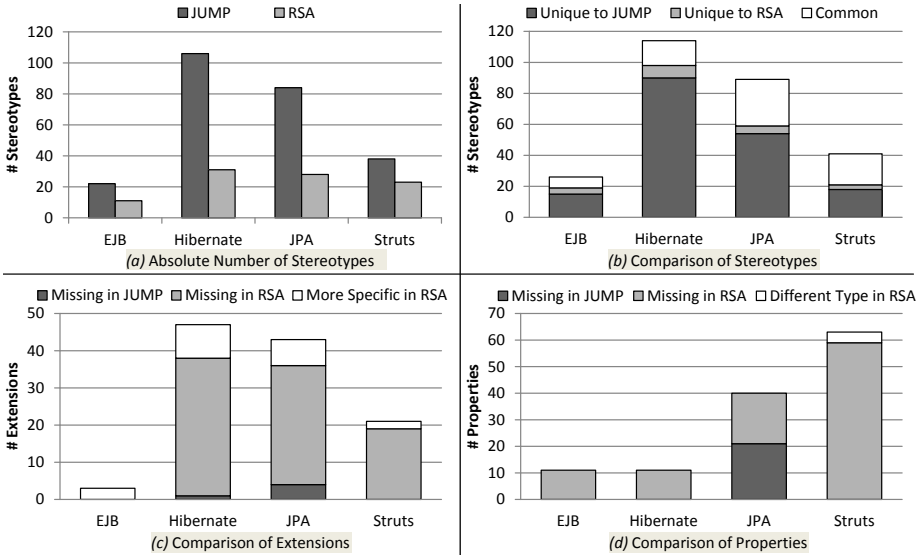

Fig. 5: Results of Quality Evaluation

additionally contains the substring "`Interface`". Another example refers to the class `QueryHint` in the JPA profile of RSA, which is in fact an annotation type in the JPA library. In our solution, the `QueryHint` is represented by a stereotype even though it is also valid to use a class instead, because the `QueryHint` can not actually be applied, but can rather only be used inside of another annotation. Although some stereotypes in the set of common ones show differences regarding the meta-classes they extend, we granted them to be equal if the extended meta-classes are related by a generalization relationship. We encountered this case in the EJB and the JPA library with respect to extensions of the meta-classes `Type` and `Class`. Stereotypes generated by *JUMP* extend the more general meta-class `Type` because the scope of Java's element type `Type` also covers `Enumeration`, `Interface` and `AnnotationType` in addition to `Class`.

The comparison regarding extensions of stereotypes common to both *JUMP* and RSA is summarized in Figure 5c. In a few cases, the RSA profiles comprise extensions to the UML meta-class `Association` to allow stereotypes on associations between elements rather than on properties contained by associations. Although both modeling variants are valid, we adhere to the second one as it is more accurate w.r.t. the target specifications of the original annotation type declarations.

Finally, in Figure 5d, the differences regarding the properties of common stereotypes are presented. Except for the JPA profile, we cover all stereotype properties of the RSA profiles. Consequently, our profiles are more complete. The main reason for missing properties in our JPA profile seems to be that RSA provides additional properties for code generation purposes, but these properties are not covered by the JPA library.

Discussion. In this study, we have demonstrated that automatically generated UML profiles from Java libraries comprise a more comprehensive set of stereotypes and features compared to profiles used in practice for the purpose of supporting such libraries. Clearly, the purpose of the developed profiles plays an important role. From a forward-engineering perspective, one may argue that the set of stereotypes, which is actually supported by the accompanying code generators is reasonable to capture at the modeling level. In fact, RSA offers code generation capabilities specific to the profiles we have evaluated in this study. However, for unsupported annotations, which have no corresponding stereotypes, code generators may only produce program code by conventions without allowing developers to intervene in this generation process at the modeling level. From a reverse-engineering perspective, we would lose relevant information at the modeling level if offered profiles provide less capabilities compared to the programming level, which is, however, the case for RSA profiles. Hence, with a fully automated approach, the quality of current profiles can be improved by providing more complete stereotypes that precisely capture the intention of the original annotation types in terms of target definitions, member declarations and return values of such members.

Threats to Validity. There are two main threats that may jeopardize the internal validity of this study. First, we consider only profiles from RSA. The main reason for this procedure is that RSA applies a similar approach as *JUMP* and offers specific UML profiles for Java libraries. Furthermore, RSA offers standard-compliant UML profiles that conform to the same UML 2 metamodel implementation as used in *JUMP*. Second, it may be possible that we missed correspondences between elements of the profiles involved in the study. Several kinds of heterogeneities [46] exist that are real challenges

for model matching algorithms and, thus, may affect the results of our study. However, by applying a two-step matching process which includes a syntactic as well as semantic comparison phase, we tried to minimize the possibility of missing correspondences as a result of different naming conventions and modeling styles. While in the first phase we used a quite conservative matching strategy to avoid false positives, we applied a rather liberal strategy in the second phase to avoid losing potential correspondences.

Concerning external validity, *JUMP* sets the focus on Java annotations. Many libraries embrace them and real-world cases provide validity for annotated Java code [39]. However, we cannot claim any results outside of Java.

6 Related Work

We investigated three lines of research: (i) mappings between Java and UML, (ii) generation of UML profiles and (iii) metamodel generation from programming libraries.

Mapping Java and UML. The elaboration on the mapping between Java and UML has a long tradition in software engineering research [15, 23, 28, 33]. Round-trip engineering for UML and Java has been extensively studied in the context of the development of FUJABA [33]. One particular concept of UML that received much attention in the context of Java code generation is the association concept [2,20,21]. However, none of these mentioned approaches consider the transformation of annotation types and their applications from Java to UML. The only exception is the mTurnpike approach [44] that considers Java annotations at the modeling level. Thereby, round-trip transformations between UML models and Java code are realized by considering stereotypes and annotations in the transformations. In contrast, *JUMP* sets the focus on the automated generation of UML profiles that facilitate round-trip transformations or transformations in general. Besides academic efforts, today's modeling tools support the transformation of Java code to UML models, and vice versa. Their current capabilities and limitations w.r.t. *JUMP* are discussed in Section 5.1.

Generating UML Profiles. The only area we are aware of approaches that deal with the automated generation of profiles, is concerned with bridging the gap between MOF-based metamodels and UML's profile mechanism, which is also related to the discussion of an external DSMLs vs. internal DSMLs in UML. Several papers discuss the pros and cons of these approaches(e.g., [42]) and their combination (e.g., [45]). The visualization of domain-specific models in UML with profiles is discussed in [22]. Abouzahra *et al.* [1] present an approach for interoperability of UML models and DSML models based on mappings between the DSML metamodel and the UML profile. Brucker and Doser [8] go one step further and propose an approach for extending a DSML metamodel for deriving model transformations able to transform DSML models into UML models that are automatically annotated with stereotypes. A related approach is presented in [47], where mappings between the UML metamodel and a DSML metamodel are defined and processed to generate UML profiles for the given DSMLs.

Generating Metamodels. To the best of our knowledge, there is only one automated approach for generating modeling languages from programming libraries—all other automated approaches that deal with exploring libraries, such as [9], set their focus on the generation of domain models rather than a language. API2MoL [11] deals

with generating metamodels based on Ecore [14] from Java APIs as well as models conforming to the generated metamodels for Java objects instantiated from the Java APIs, and vice versa. As a result, an external Domain-Specific Modeling Language (DSML) is generated from a Java API. While the general idea and motivation of the API2MoL approach is comparable to *JUMP*, there is a significant difference on how the DSML is realized. *JUMP* targets UML modelers that are familiar with UML class diagrams and generates internal DSMLs by exploiting the language-inherent extension mechanism of UML, i.e., *UML Profiles*. Furthermore, annotations are not explicitly considered in the metamodel generation process of API2MoL. One possible reason for neglecting them is that standard versions of current meta-modeling languages, such as Ecore, do not support language-inherent extension mechanisms out-of-the-box [31]. Antkiewicz *et al.* [3] present a methodology for creating framework-specific modeling languages. While we aim for an automated approach, Antkiewicz *et al.* use a manual one to create the metamodel and the transformations between model instances and instantiated objects of the frameworks. Again, annotations are not captured by the created languages. When considering the term modeling language in a broader scope, research of related fields consider ontologies as a kind of (meta-)model [19]. In particular, research on ontology extraction from different artifacts is commonly subsumed under the term *ontology learning* [13]. We are aware of only one approach for extracting ontologies from APIs [40], which neglects, however, also annotations.

To summarize, *JUMP* is—to the best of our knowledge—the first approach to generate standard-compliant UML profiles from Java libraries that exploit annotations.

7 Conclusion

With *JUMP*, we proposed an approach to close the gap between programming and modeling concerning annotation mechanisms. Thereby, we set the focus on the "Java2UML" case and demonstrated the feasibility of *JUMP* by generating high-quality UML profiles for numerous Java libraries and applied them in practical reverse-engineering and forward-engineering scenarios. The results gained by our evaluation seem promising. Still, a number of future challenges remain to further integrate programming and modeling. Some interesting differences between Java annotations and UML profiles remain to be explored. On the UML side, inheritance between stereotypes is possible, a concept that is not supported by Java for annotation types. Thus, the design quality of automatically generated UML profiles can be enhanced by exploiting inheritance. On the Java side, retention policies determine at which stages annotations are accessible. UML stereotypes are considered only at design-time. Therefore, an interesting line of future work is to support stereotype applications also during run-time, which becomes especially interesting for executable models, a research area that is currently experiencing its renaissance by the emergence of the FUML standard [34]. Furthermore, we plan to study the support of annotations in other programming languages, e.g., by investigating attributes in C# and decorators in Python, and how these concepts corresponds to UML profiles. Finally, as we set the focus in this work to platform-specific profiles, we plan to extend this scope to profiles that capture annotations independent of platforms, thereby shifting their application to a more conceptual level.

References

1. Abouzahra, A., Bézivin, J., Fabro, M.D.D., Jouault, F.: A Practical Approach to Bridging Domain Specific Languages with UML profiles. In: Proc. Workshop on Best Practices for Model Driven Software Development. pp. 1–8 (2005)
2. Akehurst, D.H., Howells, W.G.J., McDonald-Maier, K.D.: Implementing Associations: UML 2.0 to Java 5. SoSyM 6(1), 3–35 (2007)
3. Antkiewicz, M., Czarnecki, K., Stephan, M.: Engineering of Framework-Specific Modeling Languages. TSE 35(6), 795–824 (2009)
4. Bergmayr, A., Bruneliere, H., Cánovas, J., Gorroñogoitia, J., Kousiouris, G., Kyriazis, D., Langer, P., Menychtas, A., Orue-Echevarria, L., Pezuela, C., Wimmer, M.: Migrating Legacy Software to the Cloud with ARTIST. In: Proc. CSMR. pp. 465–468 (2013)
5. Bergmayr, A., Wimmer, M.: Generating Metamodels from Grammars by Chaining Translational and By-Example Techniques. In: Proc. MDEBE. pp. 22–31 (2013)
6. Brambilla, M., Cabot, J., Wimmer, M.: Model-Driven Software Engineering in Practice. Morgan & Claypool Publishers (2012)
7. Briand, L.C., Labiche, Y., Leduc, J.: Toward the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software. TSE 32(9), 642–663 (2006)
8. Brucker, A.D., Doser, J.: Metamodel-based UML Notations for Domain-specific Languages. In: Proc. ATEM. pp. 1–15 (2007)
9. Bruneliere, H., Cabot, J., Jouault, F., Madiot, F.: MoDisco: A Generic and Extensible Framework for Model Driven Reverse Engineering. In: Proc. ASE. pp. 173–174 (2010)
10. Canfora, G., Di Penta, M., Cerulo, L.: Achievements and Challenges in Software Reverse Engineering. CACM 54(4), 142–151 (2011)
11. Cánovas, J., Jouault, F., Cabot, J., Molina, J.G.: API2MoL: Automating the Building of Bridges between APIs and Model-Driven Engineering. Information & Software Technology 54(3), 257–273 (2012)
12. Czarnecki, K., Helsen, S.: Feature-based Survey of Model Transformation Approaches. IBM Systems Journal 45(3), 621–646 (2006)
13. Drumond, L., Girardi, R.: A Survey of Ontology Learning Procedures. In: Proc. WONTO. pp. 1–12 (2008)
14. Eclipse Foundation: Eclipse Modeling Framework (EMF) (2014), https://www.eclipse.org/modeling/emf
15. Engels, G., Hücking, R., Sauer, S., Wagner, A.: UML Collaboration Diagrams and their Transformation to Java. In: Proc. UML. pp. 473–488 (1999)
16. France, R., Bieman, J., Mandalaparty, S., Cheng, B., Jensen, A.: Repository for Model Driven Development (ReMoDD). In: Proc. ICSE). pp. 1471–1472 (2012)
17. France, R.B., Rumpe, B.: The Evolution of Modeling Research Challenges. SoSyM 12(2), 223–225 (2013)
18. Fuentes-Fernández, L., Vallecillo, A.: An Introduction to UML Profiles. Europ. Journal for the Informatics Professional 5(2), 5–13 (2004)
19. Gasevic, D., Djuric, D., Devedzic, V.: Model Driven Engineering and Ontology Development (2. ed.). Springer (2009)
20. Génova, G., del Castillo, C.R., Lloréns, J.: Mapping UML Associations into Java Code. JOT 2(5), 135–162 (2003)
21. Gessenharter, D.: Mapping the UML2 Semantics of Associations to a Java Code Generation Model. In: Proc. MODELS. pp. 813–827 (2008)
22. Graaf, B., van Deursen, A.: Visualisation of Domain-Specific Modelling Languages Using UML. In: Proc. ECBS. pp. 586–595 (2007)

23. Harrison, W., Barton, C., Raghavachari, M.: Mapping UML Designs to Java. In: Proc. OOPSLA. pp. 178–187 (2000)
24. Heidenreich, F., Johannes, J., Seifert, M., Wende, C.: Closing the Gap between Modelling and Java. In: Proc. SLE. pp. 374–383 (2010)
25. Jézéquel, J.M., Combemale, B., Derrien, S., Guy, C., Rajopadhye, S.: Bridging the Chasm between MDE and the World of Compilation. SoSym 11(4), 581–597 (2012)
26. Kazman, R., Woods, S.G., Carrière, S.J.: Requirements for Integrating Software Architecture and Reengineering Models: CORUM II. In: Proc. WCRE. pp. 154–163 (1998)
27. Klint, P., Lämmel, R., Verhoef, C.: Toward an Engineering Discipline for Grammarware. ACM Trans. Softw. Eng. Methodol. 14(3), 331–380 (2005)
28. Kollman, R., Selonen, P., Stroulia, E., Systä, T., Zündorf, A.: A Study on the Current State of the Art in Tool-Supported UML-Based Static Reverse Engineering. In: Proc. WCRE. pp. 22–32 (2002)
29. Kolovos, D., Di Ruscio, D., Pierantonio, A., Paige, R.: Different Models for Model Matching: An Analysis of Approaches to Support Model Differencing. In: Proc. CVSM. pp. 1–6 (2009)
30. Kurtev, I., Bézivin, J., Akşit, M.: Technological Spaces: An Initial Appraisal. In: Proc. CoopIS. pp. 1–6 (2002)
31. Langer, P., Wieland, K., Wimmer, M., Cabot, J.: EMF Profiles: A Lightweight Extension Approach for EMF Models. JOT 11(1), 1–29 (2012)
32. Lee, A.: A Scientific Methodology for MIS Case Studies. MIS Quarterly pp. 33–50 (1989)
33. Nickel, U., Niere, J., Zündorf, A.: The FUJABA Environment. In: Proc. ICSE. pp. 742–745 (2000)
34. OMG: FUML (2011), http://www.omg.org/spec/FUML/1.0
35. OMG: MOF (2011), http://www.omg.org/spec/MOF
36. OMG: Catalog of UML Profile Specifications (2014), http://www.omg.org/spec
37. Oracle: JLS7 (2013), http://docs.oracle.com/javase/specs
38. Pardillo, J.: A Systematic Review on the Definition of UML Profiles. In: Proc. MODELS. pp. 407–422 (2010)
39. Parnin, C., Bird, C., Murphy-Hill, E.: Adoption and Use of Java Generics. Empirical Software Engineering 18(6), 1–43 (2012)
40. Ratiu, D., Feilkas, M., Jurjens, J.: Extracting Domain Ontologies from Domain Specific APIs. In: Proc. CSMR. pp. 203–212 (2008)
41. Runeson, P., Höst, M.: Guidelines for Conducting and Reporting Case Study Research in Software Engineering. Empirical Software Engineering 14(2), 131–164 (2009)
42. Selic, B.: The Less Well Known UML: A Short User Guide. In: Proc. SFM. pp. 1–20 (2012)
43. UML-Profile-Store: Project Web Site (2014), http://code.google.com/a/eclipselabs.org/p/uml-profile-store
44. Wada, H., Suzuki, J.: Modeling Turnpike Frontend System: A Model-Driven Development Framework Leveraging UML Metamodeling and Attribute-Oriented Programming. In: Proc. MODELS. pp. 584–600 (2005)
45. Weisemöller, I., Schürr, A.: A Comparison of Standard Compliant Ways to Define Domain Specific Languages. In: Proc. ATEM. pp. 47–58 (2007)
46. Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schoenboeck, J., Schwinger, W.: Towards an Expressivity Benchmark for Mappings based on a Systematic Classification of Heterogeneities. In: Proc. MDI. pp. 32–41 (2010)
47. Wimmer, M.: A Semi-Automatic Approach for Bridging DSMLs with UML. IJWIS 5(3), 372–404 (2009)