APIs *à gogo*: Automatic Generation of Ontology APIs

Fernando Silva Parreiras Carsten Saathoff Tobias Walter Thomas Franz
Steffen Staab

ISWeb — Information Systems and Semantic Web, University of Koblenz-Landau
Universitaetsstrasse 1, Koblenz 56070, Germany
{parreiras, saathoff, walter, franz, staab}@uni-koblenz.de

Abstract

When developing application programming interfaces of ontologies that include many instances of ontology design patterns, developers of semantic web applications usually have to handle complex mappings between descriptions of information given by ontologies and object oriented representations of the same information. In current approaches, annotations on API source code handle these mappings, leading to problems with reuse and maintenance. We propose a domain-specific language to tackle these mappings in a platform independent way – agogo. *Agogo provides improvements on software engineering quality attributes like usability, reusability, maintainability, and portability.*

1. Introduction

Upper level ontologies and domain ontologies comprise many occurrences of a variety of ontology design patterns (ODPs) [5]. These ontologies are generally large and densely axiomatized. Therefore, the development of dedicated application programming interfaces (APIs) instead of generic solutions like RDF or OWL APIs eases the adoption of such ontologies.

When developing such dedicated APIs, developers of semantic web applications face the challenge of mapping descriptions of complex relations or entities to object oriented (OO) representations thereof. For example, core ontologies such as LCO [6], COMM [1], X-COSIMO [4] or Event-Model-F [19] represent complex objects like a multimedia annotation, a conversation among several participants or an event decomposition. Such objects are not represented by a single instance of a class but by ontology design patterns involving a number of connected (linked) instances.

The task of implementing object manipulation functionality becomes complex as well. For example, the specification of creation or deletion of multimedia objects is spread out in a number of connected (linked) data instances using decompositions, descriptions and segments.

Specifying interfaces for manipulating ontologies should provide constructs that enable to handle complex structures defined by ontologies. Accordingly, such constructs need to map from a single programming object to multiple RDF statements.

In current approaches like So(m)mer [22] or Elmo [14], annotations stored as plain text on API source code handle these mappings. These approaches have the following disadvantages:

- Low level of abstraction. When it comes to complex mappings between ontology classes and OO classes, current approaches require developers to deal with platform-specific details like database connection, data validation, deviating attention from the mappings.
- No portability. The APIs are tightly coupled to the programming language used and cannot be easily ported to other programming platforms.
- Low reuse rate. Mappings between ontology classes and OO classes are in the form of annotations. These annotations are stored as plain text and to be reused, they have to be copied instead of being referred.
- Hard maintenance. Changes of mappings on the ontology usually imply changing all occurrences of a given Java annotation, since mappings are stored as annotations and must be copied to be reused.

Indeed, addressing these issues has been one of the objectives of the field of model-driven engineering (MDE) [12], i.e., to develop and manage abstractions of the solution domain towards the problem domain in software design. Considering the expansion and usage of MDE techniques, we investigate the following problems in this paper: What MDE techniques address the aforementioned issues?

What are the results of applying these techniques in ontology API development?

Tackling the aforementioned problems results in improving software engineering quality characteristics of ontology API specifications like usability, maintainability and portability. For example, developers may concentrate on the mappings instead of taking care of problems inherent in programming. By considering mappings as first-order objects rather than as annotations, developers can keep track of mapping ontology elements like classes and properties. Finally, by introducing an abstraction from the programming language, developers may generate APIs for different programming languages or domain-specific APIs.

We introduce a model-driven approach, *agogo*, that provides a development environment for API developers to handle complex mappings, to define, and to reuse complex ODPs, and to automatically generate ontology API code. Moreover, we present results of comparing *agogo* with existing ontology API code, showing drastic reduction in size.

We organize this paper as follows: after introducing the challenges and benefits of *agogo*, we analyze current approaches in Sect. 2. We derive requirements based on our experience in developing APIs for core ontologies (COMM [1], X-COSIMO [4], Event-Model-F [19]) in Sect. 3. Section 4 presents the techniques and artifacts used by *agogo* to tackle those requirements. We describe how *agogo* uses these techniques and artifacts by example in Sect. 4.2. In Sect. 5, we analyze how the *agogo* approach allows for improving quality of ontology APIs based on the quality characteristics introduced in this section. Finally, Sect. 6 concludes this paper, pointing to future work.

2. Related Work

Developers of ontology APIs count on a variety of approaches with different abstraction levels. In the following, we analyze these approaches according to the abstraction level.

Generic solutions for developing ontology APIs are the Jena API [24] and the Sesame API [2]. However, these approaches are triple-based, i.e., developers have to work with methods such as getSubject and getObject. Problems like low abstraction level and high complexity are aggravated when dealing with big ontologies.

RDFReactor [23] and [11] are "plain" RDFS - Java/OO mapping approaches. These approaches do not provide support to complex mappings like the ones implied by ontology design patterns, i.e., developers have to program one java class for each ontology class. Moreover, when the ontology changes, developers have to manually change ontology API code.

Solutions with higher abstraction level are ActiveRDF [16], Elmo [14] and Sommer [22]. Approaches like Elmo and Sommer rely on Java annotations to specify mappings whereas ActiveRDF works exclusively for Ruby programs. As we have seen, annotations are hard to maintain and to debug. Moreover, all applications force API developers to commit to one programming language.

3. Running Example and Requirements

From the set of ontology design patterns found in the COMM ontology, we use the Semantic Annotation Pattern to illustrate the solution presented in this paper. The basic rationale applies to any other pattern used in COMM, X-COSIMO [4] and Event-Model-F [19]. Figure 1 illustrates the semantic annotation pattern as defined by the COMM ontology and the desired classes of the API in the programming model.

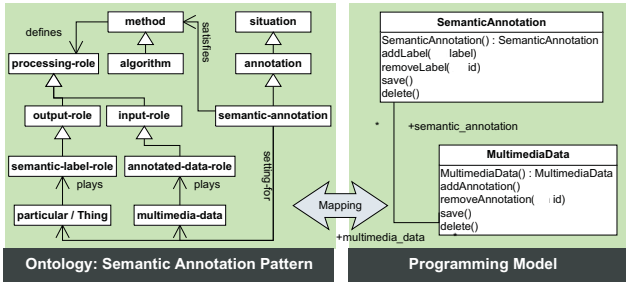

Figure 1. Ontology and API for the Semantic Annotation Pattern.

The pattern describes the annotation of a multimedia item with some label, e.g., the annotation of a part of a photo with a label pointing to a person – Carsten (not included in Fig. 1). This association is embodied through a semantic-annotation that satisfies a method (e.g., algorithms for image recognition) that defines a semantic-label-role as well as an annotated-data-role. The multimedia-data has to play the annotated-data-role, which identifies the part of the image that is annotated. The depicted particular has to play the semantic-label-role, e.g., the instance Carsten.

The COMM API maps between such patterns and Java objects. For instance, objects of the class `SemanticAnnotation` represent instantiations of the pattern semantic-annotation. The mapping is achieved by implementing the intended behavior for create, read, update, and delete operations (CRUD) that affect the knowledge base accordingly:

Create: The construction of a *new* object, i.e., an object representing data that is *not yet present* in the knowledge base, needs to result in the correct and complete instantiation of an ontology pattern.

Read: The construction of an object based on *existing* data in the knowledge base. Although very similar from an

application programming interface point of view, the underlying operation in the knowledge base is fundamentally different. In this case, the knowledge base is queried for the instance of a pattern and all involved resources and statements required to fully instantiate the object.

Update: The update of an object needs to result in the replacement of information in the knowledge base. Thereby, developers may implement different update behaviors. For example, the class MultimediaData implements a method to add a SemanticAnnotation. This method may either add a semantic label to an existing SemanticAnnotation for the image or create a new instance of a SemanticAnnotation.

Delete: The deletion of an object may have different implications. For instance, the deletion of our SemanticAnnotation may result in the deletion of the *relation* between the image and Carsten as expressed by the instance of the pattern. In another scenario, developers may want to delete the image and Carsten as well or to delete the representation of Carsten.

Commonly, solutions described in Sect. 2 are used to build ontology APIs. Based on our own experience in developing core ontologies like COMM, X-COSIMO and Event-F and their APIs, we have identified problems and derived the following requirements:

Req1. Emphasis on domain concepts. When programming ontology APIs, developers have to deal with many aspects inherent in programming languages like database access coding or data validation coding. For example, for each mapping, developers have to write code for handling access to the knowledge base. These tasks divert developers' attention from the specification of ontology APIs.

Moreover, currently, developers have to redundantly implement programming code for validating the correct instantiation of objects, e.g., code that checks whether all required information is available in an object. In our example, the Java class `SemanticAnnotation` needs to provide code that checks whether all information for a correct instantiation of the *Semantic Annotation Pattern* is available. The instantiation of this pattern without both the part of the image and the depicted person would make no sense.

Req2. Patterns as first-class citizens. Currently, when specifying standard behaviors for CRUD operations, developers have no choice but tangling the specification over the many classes that implement the pattern. Thus, developers cannot easily reuse these operations across software projects or programming languages.

Req3. Support for debugging. The ontology API code consists of several, often complex queries. Such queries are typically represented as strings and are not always recognized by programming languages or programming environments during compile time. This makes debugging particularly hard for two reasons: First, the programming environment gives no hints for syntax errors during compile time. Accordingly, developers can track syntax errors only at runtime. Second, even at runtime, semantic errors are hard to recognize. For instance, the following SPARQL-query has the correct syntax but would not return any results due to the mistyped concept name *semantic-an(n)otation*: *"select ?s where {?s a comm:semantic-anotation}"*

Req4. Change management. As the programming code references ontology concepts that the programming environment ignores, refactoring code in case of ontology changes is difficult. For instance, if a developer changes the ontology concept semantic-annotation to Annotation, associations in the programming code (e.g. annotations, query strings, URI strings) need to be updated manually.

Req5. Generation of different APIs for the same ontology or for different platforms. Currently, mappings cannot be easily reused in other programming languages since they are implemented by programming code and specific means provided by a programming language, e.g. Java annotations.

The problems that motivate these requirements impair the development of ontology APIs by retarding their availability, affecting the adoption of the respective ontologies. Moreover, having families of APIs for a given ontology or APIs for different platforms is implausible due to the effort needed.

To enforce the importance of these requirements, we analyze the current COMM API. The current COMM ontology has 702 classes while its API has 34 packages, 294 classes, 1823 functions and 11597 non-commenting source statements (NCSSs).

4. The *agogo* Approach

agogo is a model-driven approach for automatically generating OWL APIs on demand. To tackle the problems presented in the previous section, *agogo* relies on technologies regularly applied in model-driven development: metamodeling, concrete syntax and model transformations.

Agogo's metamodel and concrete syntax constitute a domain-specific language (DSL) that provides an abstraction layer over programming languages, encapsulating redundant data validation or implementation behavior. The DSL simplifies the process of specifying ontology APIs by focusing on domain concepts (Req.1).

Moreover, the usage of metamodels allows for defining concepts in a structured way, improving maintainability (Req.4). For example, elements of the ontology API specifi-

cation are maintained as single units instead of being stored in annotations.

The definition of constraints on concepts in the *agogo* metamodel improves compile-time checking, i.e., it enables API developers to validate API specifications against these constraints, minimizing errors at runtime (Req.3).

The concrete syntax for ontology API specification enables users to model patterns as first-class citizens (Req.2). For example, developers specify CRUD operations and patterns using SPARQL syntax independently from the class definition. Furthermore, the concrete syntax allows for identifying missing references and for helping to find error before code generation.

Model transformations allows for code generation to eventually more than one platform, overcoming the restriction on programming language (Req.5). Additionally, model transformations ease the creation of families of APIs. For example, developers may consider releasing a subset of the COMM API for lightweight applications.

In the next subsections, we detail the application of these techniques. Please, refer to the project web site [1] for complete specifications of the artifacts.

4.1. Key domain concepts

The *agogo* metamodel defines the concepts of a ontology API specification and corresponds to the abstract syntax of *agogo* DSL. The definition of the concepts of a ontology API specification in a metamodel raises the abstraction level and allows API developers to work exclusively with relevant constructs. For example, developers may handle concepts like mappings, patterns and operation without considering implementation issues.

In the following we describe *agogo* key concepts. Figure 2 depicts how these concepts are related in the *agogo* metamodel.

Classes. The construct Class defines the associations between platform specific classes and ontology classes. The property ontoElement associates classes to patterns or ontology classes.

Patterns. When a platform specific class does not correspond directly to a single ontology class but to an occurrence of an ontology design pattern (ODP), the concept of pattern applies. The construct QueryPattern describes ODPs using SPARQL queries [18]. It is possible to define patterns for classes, properties and operations.

Operations. CRUD operations (Create, Read, Update and Delete) are defined in ontology APIs to enable manipulation of ontology classes. These operations as well as patterns may be defined in a platform independent way by using SPARQL-like syntax.

[1] http://isweb.uni-koblenz.de/Research/agogo

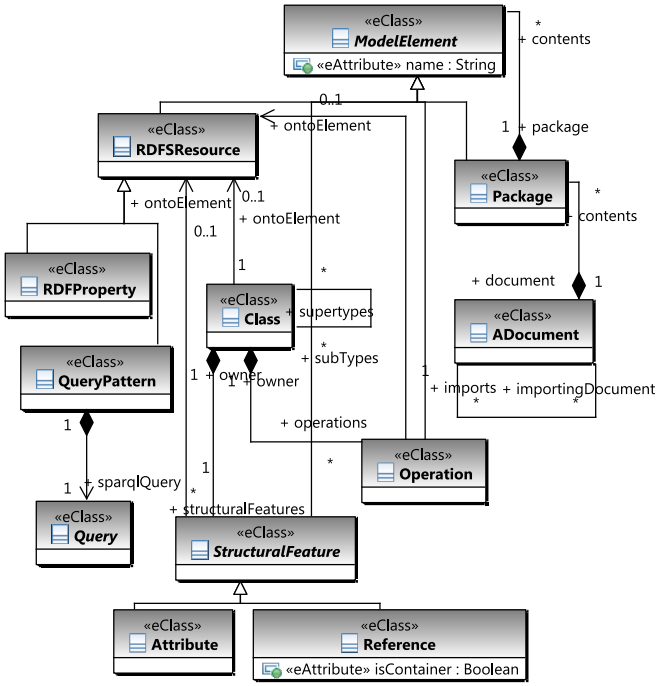

Figure 2. Snippet of the *agogo* Metamodel.

Imports. Developers may group patterns for classes, properties and operations into packages and make them available or reuse them in another API specification.

The *agogo* metamodel imports and reuses existing metamodels like the SPARQL metamodel developed in our previous work [21] and the OMG Ontology Definition Metamodel (RDF and OWL metamodels) [15].

Metamodel Constraints. Together with the *agogo* metamodel, we define constraints used by the syntax checker to enforce valid ontology API specifications. This functionality allows for identifying errors before generating ontology APIs.

Listing 1. Constraints on the *agogo* Metamodel.

```
context Operation
inv inv1:
    self.ontoElement.SPARQLQuery.whereClause
  .variables.includesAll(self.parameters);

context Property
inv inv2: self.ontoElement.SPARQLQuery
  .variables.varname.includes("obj");
```

In Listing 1, we exemplify these constraints with two OCL constraints. In the first constraint, we enforce that all

variables passed as parameter to an operation are used in the body of the query.

In the second constraint, we enforce that every pattern associated to a property must include the variable ?obj in the select statement. The predefined variable ?obj points to the range of a property in the OO representation.

4.2. *agogo* Concrete Syntax by Example

In this section, we demonstrate the main components of the *agogo* textual syntax and exemplify them with the running example. In this paper, we concentrate on how *agogo* supports patterns as first-class citizens, CRUD operations, support for debugging and change management.

To improve user experience, we have based the definition of the *agogo* textual syntax on the SPARQL syntax [18]. For example, for prefix declaration and specification of patterns, we use the SPARQL constructs.

Listing 2 presents the basic constructs of the *agogo* syntax like PACKAGE, IMPORT, CLASS and PROPERTY in exemplary fashion. We group API specifications into packages, which contain all model elements. The construct IMPORT allows for reusing classes and patterns definitions.

The construct CLASS specifies the mappings between ontology concepts and OO representations. After the class name, the reserved word TO points to a pattern declaration or directly to a SPARQL query that represents a pattern. The construct PROPERTY follows the same rationale. In Listing 2, the property label is of type dvl:particular and points to the pattern prop_label, defined in Listing 3.

Listing 2. An Example of Using Basic *agogo* Constructs.

```
PREFIX rdf:
    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX core: <http://comm/core.owl#>
PREFIX dvl:  <http://comm/dolce-very-lite.owl#>
PREFIX edns:
    <http://comm/extended-dns-very-lite.owl#>
PREFIX agogo: <http://uni-koblenz/agogo#>

PACKAGE <http://comm.agogo#> {

  IMPORT <http://comm-lite.agogo#>;

  CLASS SemanticAnnotation TO
      core:semantic-annotation{
    PROPERTY label^^dvl:particular TO
        prop_label;
    ...
```

To detach pattern specifications from class specifications, patterns must be first-class citizens, i.e., their declarations must not be associated to class declarations.

The definition of patterns is an essential point in our approach. To represent a pattern, we need to represent how ontology classes and relations compose this pattern. A user-friendly way of doing it is by using the SPARQL syntax. By using the SPARQL SELECT construct, developers describe the pattern structure.

In Listing 2, we declare that the OO class SemanticAnnotation maps onto the ontology class core:semantic-annotation and that the OO class SemanticAnnotation has a property of name label of type dvl:particular. Now, we have to specify how the values of the property label are matched. To have the labels of a semantic annotation, we need to navigate trough the structure of the Semantic Annotation Pattern (Fig. 1).

Listing 3 shows the declaration of a query pattern for the property label. The pattern is a SPARQL query that describes the structure of the Semantic Annotation Pattern. In the clause WHERE, the structure of the pattern is represented. In the clause WHERE, we have all classes and relations that need to be created, read, updated and deleted when dealing with the property label. The reader might compare the SPARQL query in Listing 3 with the classes and relations composing the pattern in Fig. 1.

The definition of patterns includes the usage of two predefined variables: ?subj and ?obj. The variable ?subj identifies the OO class, i.e., in this case, the class SemanticAnnotation, while the variable ?obj refers to the values or the property label.

For example, this pattern will match the labels associated to the class semantic-annotation, e.g., the particular Carsten (see Sect. 3). In other words, the domain of the pattern prop_label is the ontology class semantic-annotation and the range is the class particular (See declaration on Listing 2).

Listing 3. Patterns as First-class Citizens.

```
PATTERN prop_label {
  SELECT ?obj
  WHERE
    { ?subj   edns:satisfies ?method .
      ?method rdf:type       edns:method ;
              edns:defines   ?slr ;
              edns:defines   ?adr .
      ?slr  rdf:type   core:semantic-label-role .
      ?adr  rdf:type   core:annotated-data-role .
      ?obj    edns:plays     ?slr .
      ?data   edns:plays     ?adr ;
              rdf:type     core:multimedia-data .
      ?subj   edns:setting-for ?obj ;
              edns:setting-for ?data .
    }
}
```

Model transformations are responsible for generating automatically CRUD (Create/Read/Update/Delete) operations for each OO property based on the pattern specification. Although CRUD operations are generated automatically, in some cases, developers may want to customize operations. For example, developers may want to customize an insert

operation to use existing individuals.

To specify Read operations, we use standard SPARQL SELECT construct whereas to specify custom CUD operations, we use SPARQL Update [20] syntax[2]. Listing 4 shows the definition of the customized operation addLabel. The operation uses an existing instance of the class method – :method1. For each variable in the INSERT clause, one new individual is created in the ontology (except variables ?subj and ?obj).

Model transformations take specifications of CUD and generate corresponding programming language code. For example, the usage of variables (Listing 4, Lines 4-6) leads to the generation of statements to create a new instance of the class semantic-annotation-role (?slr).

Listing 4. Definition of an Operation Using SPARQL Update Syntax.

```
1  OPERATION addLabel (?obj) {
     INSERT DATA
      {
        :method1      edns:defines    ?slr.
5       ?slr          a core:semantic-label-role.
        ?obj          edns:plays      ?slr .
        ?subj         edns:setting-for  ?obj.
      }
      WHERE
10      {
         ?subj        edns:satisfies  :method1.
        }
   };
```

Developers may declare patterns anonymously, i.e., developers may associate patterns directly with properties or classes. Listing 5 shows the specification of a pattern associated with the property semantic_annotation.

Listing 5. Mapping a Property onto a Pattern.

```
1 CLASS MultimediaData TO core:multimedia-data {
   PROPERTY
       semantic_annotation^^core:semantic-annotation
     TO {
    SELECT  ?obj
    WHERE
5     {?obj      edns:setting-for  ?subj ;
             rdf:type core:semantic-annotation ;
                 edns:satisfies    ?method .
      ?method   rdf:type          edns:method ;
                 edns:defines      ?adr .
10    ?adr rdf:type core:annotated-data-role .
      ?subj     edns:plays        ?adr .
     }
   };
```

The definition of the SPARQL syntax together with the SPARQL metamodel allows for identifying non well-formed SPARQL statements. Consequently, developers may check for syntax errors at design time. Moreover, by integrating the ODM metamodel into the *agogo* metamodel, *agogo* allows for enforcing the ontology as schema for the specification. If developers mistype names of classes or individuals, the syntax checker identifies that there is no corresponding element in the ontology for that name. This functionally helps to identify typos at design time.

[2] *agogo* do not require a SPARQL Update engine. We use only the SPARQL Update syntax to generate appropriate code.

4.3. Implementation

agogo consists of a model-driven process composed of different transformations and models. Figure 3 depicts the *agogo* architecture and the embedded MDA process. Developers use *agogo* textual syntax to specify ontology API specifications. These specifications are injected to platform independent models (PIMs). We use the Atlas Textual Concrete Syntax (TCS) [9] for defining *agogo* textual syntax and Ecore [3] for defining the *agogo* metamodel.

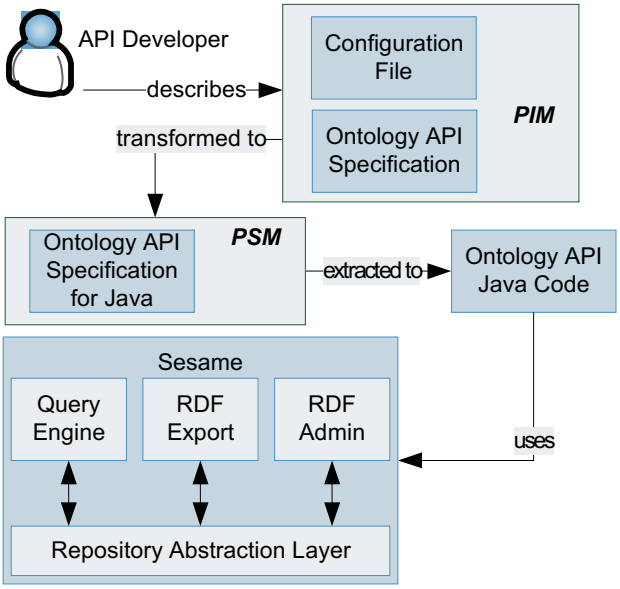

Figure 3. Architecture of the *agogo* Approach.

Model transformations take the PIM and a configuration file as input. The configuration file contains directives for the code generation like names of classes and identifiers. Consequently, model transformations produce platform specific models (PSMs) as output, which are then extracted to programming code. To specify model transformations, we use the Atlas Transformation Language (ATL) [10].

The usage of a PIM enables developers to detach the ontology API specification from programming code. Consequently, model transformations for different programming platforms may be specified, allowing code generation for multiple platforms.

We have implemented *agogo* as an eclipse plug-in. The

	agogo		Current COMM API	
	Case1	Case2	Case1	Case2
Packages	1	1	4	15
Classes	2	5	19	101
NCSS	50	70	461	3928

Table 1. Comparison of Size.

agogo DSL comprising the metamodel and the textual syntax is available for download at *agogo*'s website [3].

5. Evaluation and Discussion

In this section, we analyze how *agogo*'s functionalities affect the quality of ontology API specifications. In the following, we consider four quality characteristics of ontology API specification according to ISO 9126 [8].

Q1. Usability. One cognitive dimension of usability analysis is the abstraction level [7]. With *agogo*, developers concentrate on constructs related to the problem domain, e.g., map and pattern, raising the abstraction level.

Raising the abstraction level influences productivity. To demonstrate this impact, we have conducted an exploratory evaluation of the size of both *agogo* API specifications and Java API specifications of the running example based on the current COMM API.

As metric for size, we consider the number of non-commenting source statements (NCSSs) [17]. Table 1 summarizes the comparison of size between *agogo* and the current COMM API in two cases.

In *Case1*, we consider a specification with only two classes: SemanticAnnotation and SemanticLabel. The current COMM API requires coding 19 Java Classes and more than 400 NCSSs. With *agogo*, developers concentrate on coding 50 NCSSs in 2 classes.

To have an idea of the effort of extending or taking a subset of the COMM API, we consider the addition of the class MultimediaData in *Case2*. Although including the class MultimediaData implies implementing another ODP – the object decomposition –, the size of the ontology API increases drastically to approx. nine times the original size.

Based on this exploratory analysis, even if developers have in *agogo* half of the productivity ratio they have in Java, since the *agogo* specification is much smaller than the Java specification, the effort for producing NCSSs in Java is still higher. In other words developers are more productive with *agogo*, with benefits increasing as the API grows due to the possibilities for reuse and improved maintenance.

Q2. Reusability. By defining patterns as first-class citizens, developers may reuse patterns on different mappings. Moreover, complete libraries can be reused to generate derived APIs. For example, API developers may want to have different ontology APIs according to complexity, e.g., COMM lite and COMM full.

Q3. Maintainability. *agogo* defines constructs as metamodel concepts instead of parsing strings of text. Consequently, structured models are easier to maintain than plain text.

When the ontology changes, developers may easily change the ontolgy API specification and automatically regenerate the ontology API. The syntax checker assists refactorings like renaming, raising errors for missing references.

Moreover, constraint validation and syntax checking take place at design time, and not only at runtime as by existing approaches. The developer counts on a syntax checker for Pattern specifications.

Q4. Portability. Providing that model transformations are available, developers may easily generate an API for another language. Developers describe ontology APIs once and model transformations use the specification to generate APIs for different platforms.

agogo may be seen as an abstraction layer over existing approaches for generating ontology APIs (Sect. 2). As *agogo* does not mandate a specific programming language, developers may specify model transformations for transforming *agogo* API specifications into programming code for the platform of choice.

Nevertheless, developers need to bear in mind the effort of specifying the model transformations. To achieve abstraction from programming code, the model transformations have to handle the gap between the *agogo* API specification and the programming language. The initial effort in developing these model transformations needs to be considered when deciding to provide ontology APIs in a given programming language.

To track how the *agogo* approach addresses the requirements of Sect. 3 and affects ontology API quality characteristics, we present a traceability matrix in Table 2. It relates *agogo* requirements, the artifacts that tackle these requirements (metamodel (MM), concrete syntax (CS), and transformations (T)), examples and their relations to quality attributes. As one may notice, by establishing a domain-specific notation for designing ontology APIs, we improve the quality characteristics above, corroborating the literature on domain specific languages [13].

6. Conclusion

This paper presents a model-driven solution for designing mappings between complex ontology descriptions and Object Oriented representations – *agogo*. The solution comprises a DSL and model transformations to generate API programming code.

[3] http://isweb.uni-koblenz.de/Research/agogo

Requirement	Artifact	Example	Quality Attribute
Req1	MM, CS	Fig. 2, List. 2	Q1
Req2	MM, CS	List. 2, List. 3	Q2
Req3	MM, CS	Fig. 2, List. 1	Q3
Req4	MM, CS	List. 5	Q3
Req5	T	-	Q4

Table 2. Traceability Matrix.

agogo improves productivity on ontology API specification and enables developers with functionalities infeasible until now. Additionally, *agogo* accomplishes improvements on other quality characteristics like reusability and maintainability.

As future work, it would be interesting to investigate the usage of feature models together with ontology API specifications in validating the generation of families of APIs. For example, developers may want to generate a lite version of the API, where methods for deletion or update of ontology objects are not necessary. In this example, the feature model would be responsible for controlling the generation of these methods.

Acknowledgements

The work presented in this paper is supported by CAPES Brazil under Grant No. BEX 1553/05-4, EU STReP-216691 MOST and X-Media FP6-26978.

References

[1] R. Arndt, R. Troncy, S. Staab, L. Hardman, and M. Vacura. COMM: Designing a well-founded multimedia ontology for the web. In *Proc. of ISWC 2007 + ASWC 2007, Busan, Korea, November 11-15, 2007*, volume 4825 of *LNCS*, pages 30–43. Springer, 2007.

[2] J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In *Proc. of the ISWC 2002, June 9-12, Sardinia, Italy*, volume 2342 of *LNCS*, pages 54–68. Springer, 2002.

[3] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. J. Grose. *Eclipse Modeling Framework: A Developer's Guide*. Addison-Wesley Professional, 2003.

[4] T. Franz, S. Staab, and R. Arndt. The x-cosim integration framework for a seamless semantic desktop. In *Proc. of K-CAP '07*, Whistler, BC, 2007. ACM.

[5] A. Gangemi. Ontology Design Patterns for Semantic Web Content. In *Proc. of ISWC 2005, Galway, Ireland, November 6-10, 2005*, volume 3729 of *LNCS*, pages 262–276. Springer, 2005.

[6] A. Gangemi, M. Sagri, and D. Tiscornia. *Legal Ontologies and the Semantic Web*, chapter A Constructive Framework for Legal Ontologies. Berlin: Springer, 2005.

[7] T. R. G. Green and M. Petre. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *J. Visual Languages and Computing*, 7(2):131–174, 1996.

[8] ISO/IEC. *ISO/IEC 9126. Software engineering – Product quality*. ISO/IEC, 2001.

[9] F. Jouault, J. Bézivin, and I. Kurtev. Tcs:: a dsl for the specification of textual concrete syntaxes in model engineering. In *Proc. of GPCE '06*, pages 249–254. ACM, 2006.

[10] F. Jouault and I. Kurtev. Transforming Models with ATL. In *MoDELS Satellite Events*, volume 3844 of *LNCS*, pages 128–138. Springer, 2005.

[11] A. Kalyanpur, D. J. Pastor, S. Battle, and J. A. Padget. Automatic Mapping of OWL Ontologies into Java. In *Proc. of SEKE 2004*, pages 98–103. KSI, 2004.

[12] S. Kent. Model driven engineering. In *Proc. of the Third International Conference on Integrated Formal Methods , IFM 2002 Turku, Finland, May 1518, 2002*, volume 2335 of *LNCS*, pages 286–298. Springer, 2002.

[13] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.

[14] P. Mika and J. Leigh. Elmo, 2008. Available at `http://www.openrdf.org/`.

[15] OMG. *Ontology Definition Metamodel*. Object Modeling Group, September 2008.

[16] E. Oren, R. Delbru, S. Gerke, A. Haller, and S. Decker. ActiveRDF: Object-oriented semantic web programming. In *Proceedings of the WWW '07*, pages 817–824. ACM, 2007.

[17] R. E. Park. Software size measurement: A framework for counting source statements. Technical Report CMU/SEI-92-TR- 20, ESC-TR-92-20, Software Engineering Institute, Carnegie Mellon University, September 1992.

[18] E. Prud'hommeaux and A. Seaborne. SPARQL query language for RDF. W3C Recommendation, Jan 2008. Available at: `http://www.w3.org/TR/rdf-SPARQL-query`.

[19] A. Scherp, T. Franz, C. Saathoff, and S. Staab. A model of events based on a foundational ontology. Technical Report 02/2009, University of Koblenz-Landau, 2009.

[20] A. Seaborne and G. Manjunath. SPARQL Update: A language for updating RDF graphs. W3C Member Submission 15 July 2008, W3C, 2008. Available at `http://www.w3.org/Submission/SPARQL-Update/`.

[21] F. Silva Parreiras, S. Staab, S. Schenk, and A. Winter. Model driven specification of ontology translations. In *Proc. of ER 2008, Barcelona, Spain, October 23-26, 2008*, volume 5231 of *LNCS*, pages 484–497. Springer, 2008.

[22] H. Story. Semantic object (medata) mapper, 2008. Available at `https://sommer.dev.java.net/`.

[23] M. Völkel and Y. Sure. RDFReactor – from ontologies to programmatic data access. In *Poster Proceedings of ISWC 2005*, 2005.

[24] K. Wilkinson, C. Sayers, H. Kuno, and D. Reynolds. Efficient RDF Storage and Retrieval in Jena2. In *Proc. of the 1st Workshop on Semantic Web and Databases, Co-located with VLDB 2003, Berlin, Germany, September 7-8, 2003*, 2003.