In **Proceedings of the 3rd International Workshop on Semantic Web Enabled Software Engineering, 4th European Semantic Web Conference**, (Kendall, Elisa F. and others, Eds.), Innsbruck, Austria, KnowledgeWeb, June 2007, pp.16-30.

Enabling knowledge-based software engineering through semantic-object-relational mappings

Ioannis N. Athanasiadis[1], Ferdinando Villa[2], and Andrea-Emilio Rizzoli[1]

[1] Istituto Dalle Molle di Studi sull'Intelligenza Artificiale, Lugano, Switzerland
{ioannis,andrea}@idsia.ch
[2] Ecoinformatics Collaboratory, University of Vermont, Burlington, VT, USA
fvilla@uvm.edu

Abstract. Domain-specific conceptualizations specified as formal ontologies increase rapidly, as part of ongoing efforts for enabling the semantic web. So far, experience has shown that semantic models and their incarnations into OWL structures, though powerful for expressing complex abstractions, remain difficult to utilize in conventional software projects. In this paper we present our work for coupling ontologies with conventional domain-centric data models and object-relational persistence. The Semantic Rich Development Architecture methodology is specified for assisting the software developer to build-up enterprise applications, starting from a formal domain specification expressed in OWL, which are transcribed into object-oriented software programming interface model and relational database schema.

1 Introduction

Taking advantage of rich semantics expressed in OWL/RDF domain models in developing software projects remains a major challenge in knowledge-based software engineering. Linking object-oriented models with semantic models is required, with the goal of providing benefits for the software development process. However this linking is neither straight-forward, nor trivial, as semantic modeling and object-oriented modelling employ different conceptualizations as their building blocks for expressing domain models, as discussed in [1]. To give an example, while in O-O models classes are considered as types for instances, in OWL Classes are regarded as sets of individuals. Obviously, there is a mismatch between individuals and objects, though they both serve similar purposes. The alignment across specification languages is a challenging task, and towards this direction drives the Ontology Definition Metamodel [2] that lays the base for aligning UML models with OWL, DL and relational databases.

When Knublauch et.al [1] were presenting the Semantic Web Primer for object-oriented software developers, where the structural difference between semantic and object-oriented models are discussed, at the same time Ted Neward on his blog was characterizing object-relational mapping as the "Vietnam of Computer Science" [3]. Object-Relational Mapping (ORM) is a similar case with semantic-object mapping, that aims to put together object-orientation and

databases. In the case of ORM too, there are mismatches between the structural blocks of the two modelling paradigms. For example, in Object-Oriented programming (O-O), classes specify the characteristics (members) of object types, and there is a clear ownership of the members by the class. In the contrary, in Relational Database Schemes (RDBS) entities (tables) specify relations among things characteristics (columns); attributes are spanned among several tables, and there is a weak notion of "ownership". Typically in relational database systems, ownership is reconstructed via querying.

There have been enormous efforts from the scientific and the industrial communities to put together O-O and relational models, is order to facilitate the software development process. It is very well known that these two modelling practices don't match to each other, however each one brings along its benefits, required for enterprise application development. Using the developed mapping languages for object-relational integration, developers today are enabled, either to start from an O-O specification and inject a relational database for storing permanently their objects, or, in the vice versa, starting from a relational database, to extract programming interfaces for manipulating their content. Though some conventions have been introduced, that limit the freedom of the developers in both levels, at the very end it is a good practice that benefits from the advantages of both object programming and database technologies.

This paper, makes a step ahead, by presenting how a third layer can be introduced on top of conventional object programming and relational database models, in order to integrate semantic modeling into the software development process. The expected benefit is to enable semantic-rich software development and use of reasoning through description logics in enterprise applications. A knowledge-based software engineering environment is specified, where semantic models, object programming and relational databases are coupled smoothly (again at a price of some limitations), in order for the developer to enjoy the benefits of each technology: rich semantics and reasoning at the ontology layer, procedural behavior and state management through object-oriented programming and efficient permanent content storage in the database layer.

The rest of the paper is structured as follows; Section 2 summarizes related work on linking ontologies and databases, and object-oriented programmatic interfaces with ontologies. Based on these findings, we present in section 3 an abstract knowledge-based software engineering architecture that consists of the three aforementioned layers: semantic, object-oriented, and relational. Section 4 specifies how an ontology-based domain model can be transliterated to an object-oriented model and a relational model. Our experiences using this methodology are discussed in Section 5, and the main findings of this paper are summarized in section 6.

2 Related work

Ontologies and database cross-disciplinary efforts have so far focused in two directions: 1) persistent storage of newly created knowledge bases, and 2) popu-

lating ontologies with instances initially stored in relational databases. Various software tools and libraries exist to enable (1), e.g. Jena [4], Protégé [5] and KAON [6]. All of these allow storing OWL/RDF content in databases, making no difference between storage of OWL/RDF Classes and Individuals. Both the conceptual specification and the actual content (i.e. instances) of a semantic model are made persistent following a native 'triple-store' approach (i.e. in the form of subject-predicate-object tables). This design choice is suitable for accessing and storing ontology-specified content and is optimal for reasoning on the knowledge base [7]. Yet, it is quite inefficient for accessing and querying individuals following traditional database techniques and very cumbersome to build enterprise software applications upon. However in the Semantic Web era, not only reasoning on concepts will be necessary, but also reasoning at the instance level and efficient instance retrieval [8]. Therefore, alternative kinds of OWL instances storage that are optimized for indexing and retrieval are required. Roldan Garcia and Aldana-Montes [8] propose alternative storage models based on generated relational database schemata.

Tools are also available to populate ontologies with content from existing databases. E.g, the D2R translator and server [9, 10] enables mapping an existing database schema to RDF structures, which can be made available through a web server, and also supports querying. The Protégé DBOM plugin [11, 12] can be used for the same purpose. The Dartgrid toolkit, which won the Best Paper Award at the "Semantic Web in Use" track at the 2006 International Semantic Web Conference, is also worth mentioning. Dartgrid is an application development framework including a set of semantic tools that facilitate the integration of heterogeneous relational databases using semantic web technologies [13]. Finally, Musa-K is another ontology-mediated database querying platform [14] that employs advanced semantics for integrating sparse data sources.

Several toolkits are available for translating OWL structures into Java classes for supporting coding support for semantic applications, apart the ontology development toolkits mentioned above. One of the first translators was the Protégé Bean Generator [15], which transforms conventional frame-based Protégé ontologies into Java source code for developing JADE agents [16]. Also, Protégé-OWL incorporates code generation plugins that export Java source code following the Eclipse Modelling Framework (EMF), the Kazuki or the Java Beans conventions; cf. [17]. RDFReactor [18] is a toolkit for dynamically accessing an RDF model through domain-centric methods (getters and setters). A more sophisticated approach was presented by Kalyanpur et.al. [19], that deals with issues as multiple inheritance. However, both of them still store the generated instances as triple-stores.

3 Towards a knowledge-based software engineering architecture

In Knublauch's vision of ontology-driven software development, [20], it is underlined that that runtime access to ontologies has advantages (related to the exe-

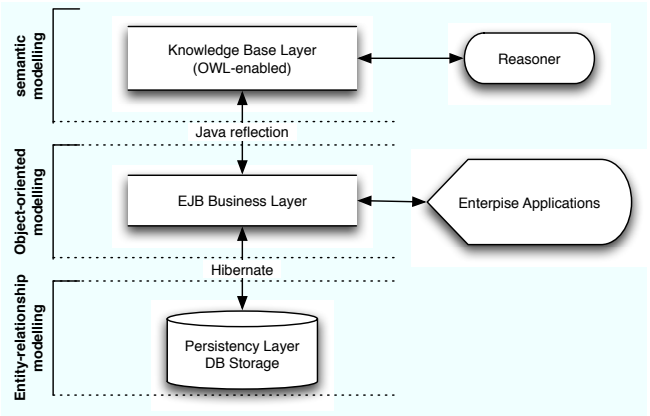


Fig. 1. The semantic-rich programming platform architecture

cution of reasoners), however it should be combined with object-oriented source code generated from OWL, so that ontology-defined structures can be smoothly integrated with object-oriented code. Going a step further, Knublauch envisions a programming practice that instead of relying on UML defines the domain model in OWL. Towards this direction drives the *Se*mantic-*Ri*ch *D*evelopment *A*rchitecture (*SeRiDA*) presented here, that combines object-oriented programming and relational databases, with semantic models. SeRiDA objective is to synthesize pertinent tools for each task: OWL models for expressing rich semantics and connecting to an external reasoner for logical operations, Enterprise Java Beans for end-user application development, and normalized relational databases for content persistence. These three layers can be combined all together in a semantic-rich development architecture illustrated in Fig. 1.

The SeRiDA enterprise application development environment employs a conventional object-oriented domain modelling practice, as Enterprise Java Beans specifications [21], for developing server-client software modules. Data are stored persistently in a relational database, through object-relational mapping, as for example those provided by EJB 3.0 Persistence or Hibernate [22]. On top of these two layers, a semantic layer is added that feeds the persistent objects into a knowledge manager and through it to a reasoner for applying logical operations. In SeRiDA we identify two modes of operation explained below.

In the first mode, we start from the top level of semantic modeling. Let an ontology be given, that specifies the conceptualization of a domain. Part of this semantic model specifies the (concrete) concepts involved, and can be translated into data structures and entities specifications, while a second part defines the logics that pertains to these concepts. Using some conventions presented in the following section, from an ontology-specification, we can derive to an object model, that can be used for programming enterprise applications, which in turn can be mapped into a relational storage. This means that based on the structures defined in the domain ontology, we can generate both the programming

interface source code and a persistence storage in a database. Having generated both the programming interface and the relational schema, the platform enables a semantic-rich framework for software development, employs OWL instead of UML and turns out with a object oriented domain model and a database schema for adding behavior to individuals, storing then in relational databases and developing client applications.

In the second mode, by keeping track of the original OWL Classes used for generating the programming interface, we can connect the Java objects at runtime back to the semantic layer (e.g. through Java reflection) and apply a reasoner on them. So for example, we can classify an object of a generated Java class according to logical definitions that were not present when source code was generated. In this way, logical specifications of classifications specified in a domain ontology using description logics can be considered as an upper layer for storing part of the business intelligence that can be updated at runtime, without affecting the conventional APIs for software coding and application development.

The main expected benefit from the SeRiDA architecture is to enable semantic-rich operations. Then comes the added value of using standard APIs for end-user application development. End-user applications can be smoothly integrated with industrial standard technologies, that do not need to be modified for enabling semantic rich computing. Client applications can be developed on top of the O-O layer, without any transition costs. Similarly, legacy data storages do not need any additional effort for becoming available in a semantic aware applications (for example to be "triplized"), apart from the conventional object-relational mapping that that is an accepted industrial standard procedure. However, there is a price to be paid by using this architecture. The major limitation is that, we keep detached ontology classes from individuals. This implies that a reasoner can be used for unidirectional inferences (from individuals to classes). For example, we can query the reasoner for the logical definitions satisfied by a certain individual, rather than asking for all inferred individuals of a certain class. Though the latter is not prohibited by this architecture, it is definitely a costly operation which a native "triplestore" storage should be preferred in large scale projects. On the contrary, we benefit from a handy object-oriented domain model, that can be used directly for enterprise application development.

4 Aligning Semantic-Object-Relational models

Though it is known that the notion of an individual in an ontology is semantically different from the definition of a class instance in object-oriented programming [19], and a tuple in a relation, we have pointed out the added value of using standard APIs for end-user application development. Here we show how from an OWL ontology we may generate a programming interface using Enterprise Java Beans with Hibernate object-relational mappings for database persistence, while taking into account important aspects of OWL as inverse properties and anonymous classes. The approach introduced below is presented in alignment

with the object-relational mapping specifications as adopted by EJB 3.0 [21] and realized by Hibernate [22, ?].

4.1 From OWL Classes to Java Classes and Entities

An OWL Class specifies both the structural features of a concept and the logical constraints that it should respect. Therefore, in a persistent storage of individuals it is not required to accommodate all classes in an ontology, rather those that define the structural extensions. It is quite common to have OWL classes that do not assign additional data properties to their parent class, but only specify restrictions. These classes are mainly intended for defining classifications for categorizing the instances of the father class. Similarly, we may have a class that inherits more than two classes, i.e. simply defining a union of several ancestors. Such union classes include the anonymous classes used to define owl:Property domains and ranges. Both these ontology class patterns are consider part of the "business" logic of the ontology, and do not contribute axioms of relevance to the concrete classes and the relational model.

We consider eligible for persistent storage only those non-anonymous OWL classes that contribute with additional attribute specifications in the class inheritance. Each of these classes extend the structural specification of the semantic model and is considered to represent an entity in the relational model, which ultimately can be assigned to a database table. To give an example, for an OWL Class `cs:Person` defined as `<owl:Class id="cs:Person">` a unique relation is defined for persisting its instances: `csPerson=(id)`, where `id` is the unique identifier of each stored instance, therefore a primary key of table `csPerson`. Along with this table comes an Entity Java Bean class that contains a single member: the `id`. Both the Entity class `Person`, and table `csPerson` will be extended with attributes and relations that derive from the OWL Properties of Class `cs:Person`.

By introducing the persistence layer, we overcome the obstacle of the "identifyability" of the objects, which was pointed out in [1]. In contrary with conventional objects, where class instances reside only in memory, and are no longer available after the code execution, using a persistent storage in SeRiDA, objects are stored permanently in a database that assigns a unique identifier (the `id`). Therefore, instances are traceable in the long run through a unique URI that combines the database with the table and the id. For example it could be something like the following URI: `jbdc:postgresql://someserver/Database#csPerson:12`).

Next comes the transliteration of OWL properties into entity attributes and Java Class members. Properties in OWL can be (a) Literal properties and (b) Object properties. Literal properties (defined through `owl:DatatypeProperty`) define data attributes of an entity, while object properties (`owl:ObjectProperty`) assign relations among tables. In the following we present how OWL properties can be mapped to table attributes and relations, with respect to OWL specific features as inverse properties, that will be discussed in par. 4.3.

4.2 Literal properties

Literal properties simply specify data type attributes of an entity, which can be easily transformed among the three layers. For example a literal of type `&xsd;string` in OWL, defines a member of type `String` in a concrete class, or a `VARCHAR` attribute of an entity. ODM provides with such a full list of these alignments.

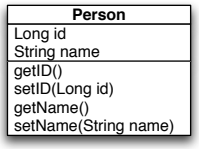
We identify two cases, depending on the cardinality constraint of the property. Note that there in OWL there are two ways of specifying the cardinality constraint of a property, as OWL properties are primal structures, independent from OWL classes. The cardinality can be defined universally at the OWL Property level, by assigning it to be a functional or non-functional property. Or it can be specified at the class level, having a cardinality constraint attached to the OWL class. In the followings functional and single-cardinality properties are treated in the same way, as in object-oriented programming properties are owned by classes.

I-a: Functional or single-cardinality literal property. A literal property with a maximum cardinality restriction equal to one defines a unary attribute within the relation of the entity (i.e. the table of the OWL Class). Following the above example of class `cs:Person`, the functional property called `cs:name`:

```
<owl:DatatypeProperty rdf:ID="name">
    <rdf:type rdf:resource="&owl;FunctionalProperty"/>
    <rdfs:domain rdf:resource="#Person"/>
    <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>
```

is assigned as an attribute to the `Person` relation:
`csPerson=(id,name)`.

The specified JavaBean would look as the following one, exposing a member `name` of type `String`.



I-b: Multiple cardinality literal property. A literal property of multiple cardinality identifies a multi-valued attribute of an entity, dependent only upon the primary key. Multi-valued attributes in normalized database systems are implemented is separate tables as associate entities, through an one-to-many relationship. In the `cs:Person` example, lets include a literal property `cs:phone` (without any additional cardinality constraints) as:

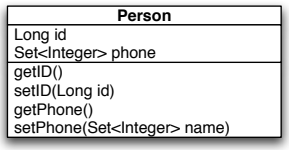
```
<owl:DatatypeProperty rdf:ID="phone">
    <rdfs:domain rdf:resource="#Person"/>
```

```
    <rdfs:range rdf:resource="&xsd;int"/>
</owl:DatatypeProperty>
```

In this case, an associated table is needed for storing the list of phones of each person through an one-to-many relationship.

```
csPerson=(id,name)
csPerson_phone=(id,phone), fkey: id=csPerson.id
```

At the programming level the phones of a person will be accessed through the `phones` member, which is a set of integers, as in the the following JavaBean:



4.3 Object properties

OWL object properties specify relationships among OWL classes, which can be translated into relationships between entities in the relational schema. We identify two kinds for object properties in OWL, based on whether there is an inverse property defined or not. The issue of being or not an inverse property emerges, as the in object-oriented programming properties are owned by classes, and the updating of inverse relations need to be specified in the source code. Also, note that there is a semantic difference in the definition of a functional property in OWL and in relational databases: in OWL, a functional property implies a universal (cross-concept) cardinality constraint equal to one. In a relational database, a functional field implies that a value is required for each tuple (i.e should not be null). In the following, we consider functional OWL properties as properties with a singular cardinality constraint.

Non-inverse object properties

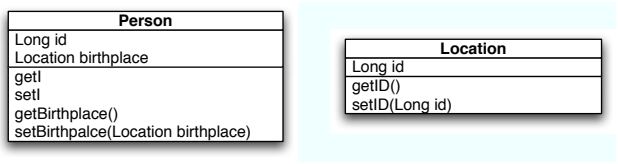
II-a Non-inverse functional (singular cardinality) object property. An non-inverse functional of singular cardinality object property can be translated as one-to-one unidirectional relationship, which is added as an attribute in the owning entity. For example, let `cs:Person` have a functional property `birthplace` with range of type `cs:Location`, as shown below:

```
<owl:ObjectProperty rdf:ID="birthplace">
    <rdf:type rdf:resource="&owl;FunctionalProperty"/>
    <rdfs:domain rdf:resource="#Person"/>
    <rdfs:range rdf:resource="#Location"/>
</owl:ObjectProperty>
```

This property can be stored in a relational schema as:

```
csPerson =(id,birthplace) fkey:birthplace=csLocation.id
                                csLocation=(id,...)
```

Using the `Person` JavaBean shown below, the birthplace property can be accessed as:



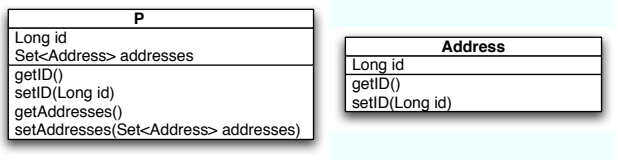
II-b Non-inverse object property. In this case, an non-inverse multi-cardinality object property defines a many-to-many unidirectional relationship between two entities. For example, an object property `hasAddress` may associate each `cs:Person` class to several `cs:Address` classes (we assume here that the `cs:Address` is not aware of its inhabitants).

```
<owl:ObjectProperty rdf:ID="hasAddress">
    <rdfs:domain rdf:resource="#Person"/>
    <rdfs:range rdf:resource="#Address"/>
</owl:ObjectProperty>
```

In a normalized database, this relation can be implemented using an intermediate relationship table, as:

```
csPerson =(id,...)
csAddress=(id,...)
csPerson_hasAddresses=(person,address) fkey: person=csPerson.id,
                                             address=csAddress.id.
```

These tables are accessible through Hibernate using the `Person` and `Location` JavaBeans:



Inverse object properties

Using the `owl:inverse-of` declaration we can specify relationships among OWL classes, that are bi-directional, i.e. can be accessed by both entities involved. This does have an impact on the schema of the database, has implication on the cascading of commands, as delete, insert and so on, and for the programming interface. Here we identify three kinds of relationships:

II-c Functional (singular cardinality) property inverse of a functional (singular cardinality) property. specifies an one-to-one bidirectional association. This can be implemented similarly with one-to-one unidirectional association in the DB level. However now it specifies one property in each Java class, i.e. there are two entry-points to this piece of information. To give an example, let's imagine that each `cs:Person` is able to own up to one `Cat`. We specify two properties `cs:owns` and `cs:hasOwner`, as:

```
<owl:ObjectProperty rdf:ID="owns">
    <rdf:type rdf:resource="&owl;FunctionalProperty"/>
    <rdfs:domain rdf:resource="#Person"/>
    <rdfs:range rdf:resource="#Cat"/>
    <owl:inverseOf rdf:resource="#hasOwner"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasOwner">
    <rdf:type rdf:resource="&owl;InverseFunctionalProperty"/>
    <rdfs:domain rdf:resource="#Cat"/>
    <rdfs:range rdf:resource="#Person"/>
    <owl:inverseOf rdf:resource="#owns"/>
</owl:ObjectProperty>
```

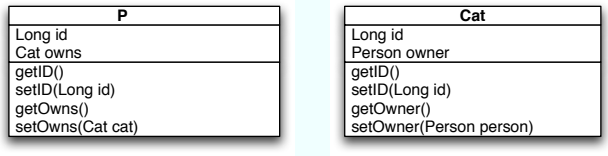
In this example,the `owl:FunctionalProperty` construct is used for OWL code simplicity, instead of a cardinality constraint. The content of both `cs:owns` and `cs:hasOwner` properties can be peristed as an extra attribute of any of the two entities, as:

```
csPerson =(id,...)
csCat=(id,person,...) fkey: person=csPerson.id
```

Or alternatively can be realized as an associate entity table which has as a primary key a unique combination of person and cat ids.

```
csPerson =(id,...)
csCat=(id,...)
csPerson_owns=(person,cat) fkey: person=csPerson.id
                                 cat=csCat.id
```

Either of the two equivalents is the DB schema, the `Person` JavaBean will have a member called `owns` that will refer to a `Cat` object and the vice versa, as:



II-d Functional (singular cardinality) property inverse of a non-functional property. It specifies a bi-directional one-to-many relationship, that can be implemented with an associate entity table, which has a primary key only the id of the entity at the singular side of the relationship.

In the same example as in the previous case, let's allow each `cs:Person` to own several cats, while each `cs:Cat` has only one owner. In OWL this relationship is expressed as:

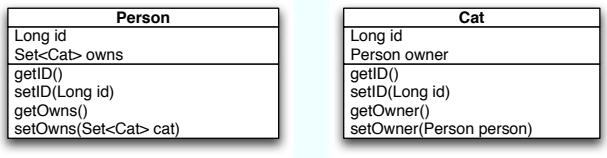
```
<owl:ObjectProperty rdf:ID="owns">
    <rdfs:domain rdf:resource="#Person"/>
    <rdfs:range rdf:resource="#Cat"/>
    <owl:inverseOf rdf:resource="#hasOwner"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasOwner">
    <rdf:type rdf:resource="&owl;FunctionalProperty"/>
    <rdfs:domain rdf:resource="#Cat"/>
    <rdfs:range rdf:resource="#Person"/>
    <owl:inverseOf rdf:resource="#owns"/>
</owl:ObjectProperty>
```

In a relational schema we need have an associate entity table to store this relation that has as primary key only `person=csPerson.id`, for expressing an one-to-many relationship as:

```
csPerson =(id,...)
csCat=(id,...)
csPerson_owns=(person,cat) fkey: person=csPerson.id
                                 cat=csCat.id
```

In the object-oriented layer level, the `Person` class has an member `owns` that refers to a set of `Cat` objects, and the `Cat` class has a member `owner` of type `Person`.



II-e Object property inverse of a object property. In the generic case that there are not any cardinality restrictions present, two inverse object properties define a many-to-many bidirectional relationship. Following the previous example of Person and Cats, lets assume that a Person may own many Cats and a Cat could have several owners. This is expressed in OWL as:

```
<owl:ObjectProperty rdf:ID="owns">
    <rdfs:domain rdf:resource="#Person"/>
```

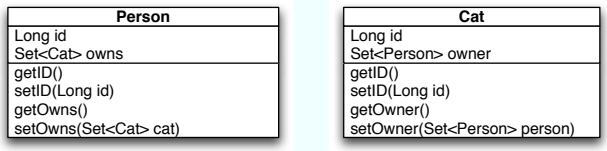
```
    <rdfs:range rdf:resource="#Cat"/>
    <owl:inverseOf rdf:resource="#hasOwner"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasOwner">
    <rdfs:domain rdf:resource="#Cat"/>
    <rdfs:range rdf:resource="#Person"/>
    <owl:inverseOf rdf:resource="#owns"/>
</owl:ObjectProperty>
```

The many-to-many relationship is stored in a normalized database through an associate entity table that has foreign key references to both `person=csPerson.id` and `cat=csCat.id`, as:

```
csPerson =(id,...)
csCat=(id,...)
csPerson_owns=(id,person,cat) fkey: person=csPerson.id
                                    cat=csCat.id
```

Finally, through object-relational mapping these tables can be accessed by `Person` and `Cat` JavaBeans that expose the members `Set<Cat>` and `Set<Cat>` respectively.



5 Implementation with EJB 3.0 and Hibernate

5.1 Inheritance and classifications

Another very important issue that OWL conceptualizations, programming domain models and relational databases do not fit together is the issue of inheritance and association. Current software development practice favors composition versus inheritance, for enabling software maintenance, code extensibility and reuse. On the contrary, in OWL, and especially OWL-DL, inheritance is preferred for favoring logical operations. Also, in object-oriented modelling inheritance through subclassing is restricted to one ancestor, while OWL allows multiple ancestors. This obstacle can be overcome at the object-oriented layer by implementing IS-A kind of relations by using Java interfaces (as each java class can `implement` several interfaces).

The relational model does not support any sort of polymorphism [3]. In object-relational mapping standards [21, **?**], object inheritance is mapped in three ways: (a) through a single table, that contains the attributes of all subclasses,

(b) using join tables for each class, where the ancestor attributes are stored in a single table for all entities, and (c) by assigning one table per class, where each entity duplicates the common attributes. In each approach there are pros and cons, that have been discussed in detail in the object-relational mapping community. In the SeRiDA framework, all three options are enabled, as this feature depends on the ORM strategy suitable for each application. This choice has no impact on the the framework, as it accesses the content through the persistent objects.

5.2 Source code and Object-relational mapping generation

Using the conventions detailed above, we implemented a software tool that starting from an OWL ontology generates the Entity Beans following the 3.0 Enterprise Java Beans specifications [21]. Also for each entity, a Hibernate object-relational mapping is generated using the aforementioned conventions. Code generations is performed through an one pass traversing through the OWL graph, where each OWL class is visited only once. For persisting a single OWL class in the database, all the Classes associated through object properties have to be mapped as well.

In our implementation we favoured the Hibernate option for specifying the object-relational mappings using XML instead of EJB annotations, as this provides a clear separation between the relational and the object-oriented layers, and allows the potential of adapting the ORM mappings to existing DBs without changing the Java code.

These issues are documented in the prototype that will become available on the Internet[3]. The translator is a plugin for the integrated knowledge management toolkit ThinkLab[4], which provides a simplified interface for programming with Protege and Jena packages. A future version of the translator may be developed as a Protege plugin.

5.3 Experinces with SeRiDA methodology

The SeRiDA methodology is currently evaluated in the software development life-cycle of the Seamless-IP EU-funded project. Seamless-IP project aims to link agronomic models and environmental data across scales and disciplines, through the Seamframe framework [23]. In Seamframe, several environmental models are required to be linked together, each one of which exposes its domain conceptualizations using ontologies. On top of this framework a client-server end-user application is being developed using Abode Flex for enabling the end users to parametrize models and visualize results. Our experiences in this context are encouraging, as in the agile application development process is has improved significantly through the generation of the Enterprise Java Beans and object-relational mappings sources. Also we experienced the advantages of having a

[3] `http://imt.svn.sourceforge.net/svnroot/imt/Thinklab/`
[4] `http://www.integratedmodelling.org`

single entry point to the data types specification of the application, i.e. the ontologies, that fosters the collaboration of people with different backgrounds (database experts, programmers, environmental modellers). A demonstration applied in modelling farming systems and management alternatives of a farm household is also presented in [24].

6 Discussion

In this paper we specified a methodology, called SeRiDA, for enabling a three-tier mapping along ontologies, object-oriented Java beans and relational databases. Though there are certain limitations, due to the conceptual mismatches among the three modeling paradigms, there a certain advantages for the software development process that can benefit from adding a semantic layer on top of the existing object-relational architectures. Through the methodology presented, starting from a domain conceptualization expressed in OWL both the object-oriented and the relational models can be derived, as semantic models are richer, containing all the information required for such an activity. Also, we have implemented a prototype that generates programming interfaces as Enterprise Java Beans and Hibernate object-relational mappings from OWL ontologies, and we are currently using it for the supporting the software life-cycle in the Seamless-IP European Project. Future efforts will concentrate on issues related to performance testing of the SeRiDA architecture in a large-scale, data intensive application, as that one of Seamless-IP.

Acknowledgements Authors would like to thank David Huber (AntOptima SA) for his comments related to the practical applications of this work. This publication has been partially funded by the EU 6th FP IP SEAMLESS (SUSTDEV-10036), and the NSF award DBI-0640837 (ARIES). Also we express our gratitude to the two anonymous reviewer for their valuable comments.

References

1. Knublauch, H., Oberle, D., Tetlow, P., Wallace, E., Pan, J.Z., Uschold, M.: A semantic web primer for object-oriented software developers. W3C Working group note, W3C (2006)
2. OMG: Ontology definition metamodel OMG Document ad/2005-08-01, OMG (2005)
3. Neward, T.: The vietnam of computer science. Blog post, The Blog Ride, (2006) http://blogs.tedneward.com/2006/06/26/The+Vietnam+Of+Computer+Science.aspx
4. McBride, B.: Jena: A semantic web toolkit. IEEE Internet Computing **6**(6) (2002) 55–59
5. Horridge, M., Knublauch, H., Rector, A., Stevens, R., Wroe, C.: A practical guide to building owl ontologies using the Protégé-OWL plugin and CO-ODE tools. Online tutorial, The University Of Manchester and Stanford University (2004)
6. KAON: The Karlsruhe ontology and semantic web framework developer's guide for kaon 1.2.7. Technical report, University of Karlsruhe, Germany (2004)

7. Wilkinson, K., Sayers, C., Kuno, H., Reynolds, D.: Efficient rdf storage and retrieval in Jena2. In: VLDB Workshop on Semantic Web and Databases. (2003) 131–150
8. Roldan Garcia, M.d.M., Aldana-Montes, J.F.: A tool for storing owl using database technology. In Grau, B.C., Horrocks, I., Parsia, B., Patel-Schneider, P., eds.: First Int'l Workshop on OWL Experiences and Directions, Galway, Ireland (2005)
9. Bizer, C.: D2R map: A database to RDF mapping language. In: Twelfth International World Wide Web Conference (WWW2003), Budapest, Hungary (2003)
10. Bizer, C., Cyganiak, R.: D2R-server:publishing relational databases on the web as SPARQL-endpoints. In: 15th International World Wide Web Conference (WWW2006), Edinburgh, UK (2006)
11. Curé, O., Squelbut, R.: Semantic mapping to synchronize data and knowledge bases at the instance level. In: European Semantic Web Conference. (2006)
12. Squelbut, R., Curé, O.: Integrating data into an owl knowledge base via the dbom protege plug-in. In: 8th Intl. Protégé Conference. (2006)
13. Chen, H., Wang, Y., Wang, H., Mao, Y., Tang, J., Zhou, C., Yin, A., Wu, Z.: Towards a semantic web of relational databases: A practical semantic toolkit and an in-use case from traditional chinese medicine. In Cruz, I.F., et.al, eds.: 5th International Semantic Web Conference. Lecture Notes in Computer Science (4273), Springer (2006) 750–763
14. Moreno, N., Navas, I., Aldana, J.: Putting the semantic web to work with db technology. IEEE Technical Committee on Data Engineering Bull. **26**(4) (2003) 49–54
15. van Aart, C., Pels, R., Caire, G., Bergenti, F.: Creating and using ontologies in agent communication. In Cranefield, S., et.al, eds.: Ontologies in Agent Systems, AAMAS, Bologna, Italy (2002)
16. Bellifemine, F., Caire, G., Poggi, A., Rimassa, G.: JADE-A white paper. EXP in search of innovation **3**(3) (2003) 6–19
17. Sharma, D.K., Johnson, T.M., Solbrig, H.R., Chute, C.G.: Transformation of protégé ontologies into the eclipse modeling framework: A practical use case based on the foundational model of anatomy. In: 8th Intl. Protégé Conference, Madrid, Spain (2005)
18. Völkel, M.: RDFReactor - from ontologies to programmatic data access. In: International Semantic Web Conference (ISWC). (2005)
19. Kalyanpur, A., Pastor, D.J., Battle, S., Padget, J.: Automatic mapping of owl ontologies into java. In: 16th Int'l Conference on Software Engineering and Knowledge Engineering, Banff, Canada (June 2004)
20. Knublauch, H.: Ramblings on agile methodologies and ontology-driven software development. In: Workshop on Semantic Web Enabled Software Engineering, International Semantic Web Conference, Galway, Ireland (2005)
21. DeMichiel, L., et al.: Enterprise javabeans 3.0 specifications. JSR 220, JCP (2006)
22. JBoss: Hibernate reference documentation. Online documentation, JBoss (2006)
23. Rizzoli, A., Athanasiadis, I., Donatelli, M., Huber, D., Muetzelfeldt, R., van Evert, F., van den Broek, M., van der Wal, M., Villa, F.: Overall architectural design of SeamFrame. SEAMLESS-IP Report 7, (2005)
24. Athanasiadis, I.N., Janssen, S., Huber, D., Rizzoli, A.E., van Ittersum, M.: Semantic modelling in farming systems researchInformation Technology in Environmental Engineering, Springer-Verlag (2007) 417–432