In **Workshop on Ontology-Driven Software Engineering at OOPSLA**, Orlando, FL, USA, 2009.

Domain object modeling with relational persistence for idiomatic OWL/RDF

Ioannis N. Athanasiadis and Andrea-Emilio Rizzoli

Dalle Molle Institute for Artificial Intelligence, USI—SUPSI, Lugano, Switzerland
ioannis,andrea@idsia.ch

Abstract. This paper investigates how to use an idiomatic OWL/RDF model as a specification language for delivering Domain Object Model with relational persistence. It presents a systematic translation of a subset of OWL/RDF constructs to object structures with a relational database back-end. The presented framework has been developed as a plugin for the Protégé ontology editor, and it has been evaluated against a benchmark of semantic repositories with promising results.

1 Introduction

The adoption of formal semantics for domain modeling has lately became very popular. Semantic models, typically expressed using the Resource Description Framework (RDF) or the Web Ontology Language (OWL), emerge for specifying several domains including earth sciences, biology, medicine, or multimedia, to name a few. Initially, semantic models were meant to be used as classifications or thesauri. Nowadays, they are the foundations for realizing Semantic Web Applications, where data, services, and people interact seamlessly over the open web. However, experience has shown that semantic models and their incarnations into OWL/RDF structures, though powerful for expressing complex abstractions, remain difficult to utilize in conventional software projects.

A major challenge for knowledge-based software engineering is to take advantage of the rich semantics expressed in OWL/RDF domain models. The goal of providing benefits for the software development process requires aligning object-oriented models (OO) with semantic models. However this linking is neither straight-forward, nor trivial, as semantic modeling and object-oriented modeling employ different conceptualizations as their building blocks for expressing domain models, as discussed in [1]. While in object models classes are considered as types for instances, classes in OWL are regarded as sets of individuals. Obviously, there is a mismatch between semantic models and object models, though they both serve similar purposes. The alignment across specification languages is a challenging task, and in this direction aims the recently published as an OMG standard Ontology Definition Metamodel [2] that lays the base for aligning UML models with RDF, OWL and topic maps.

This work investigates how a domain specification, expressed in OWL/RDF can be used (and reused) in software engineering. This is a step to enable

knowledge-based software engineering, where semantic models can serve as a domain specification language, and builds upon our prior work [3, 4]. In the following Section 2 we discuss the differences between semantic, object and relational models. Section 3 drafts a methodology for transliterating semantic models to object models with relational persistence. Section 3.4 introduces an implementation of the presented method as a plugin for the Protégé knowledge editor[1]. Finally in Section 4 we evaluate the efficiency of the proposed method in responding to queries using a benchmark for semantic repositories. The paper concludes with a discussion and open issues for future research.

2 Semantic-object-relational model impedance mismatch

Object-relational impedance mismatch is the difference resulting from the fact that relational theory is based in relationships between tuples that are queried, whereas the object paradigm is based on relationships between objects that are traversed [5]. Similarly, semantic-object impedance mismatch is the difference resulting from the fact that the object paradigm is based on static definitions of classes, whereas the semantic paradigm defines classes as a set of axioms which are logically verified. There are certain incompatibilities between semantic and object paradigms discussed in [6, 2] and between semantic and relational paradigms, discussed in [7]. The key differences between the three paradigms can be summarized as follows.

Open vs Close World Assumption: Semantic models adopt the open world assumption, while Object and Relational models adopt the closed world one. In OWL, the absence of a statement does not imply that the statement is false. On the contrary, in OO languages and SQL, the absence of a statement is considered as a false statement. For example in Java, the method call `professor.coursesTaught.contains(ECE123)` will return false if the set containing the courses taught by the object `professor` does not explicitly include the course `ECE123`. The same operation in an OWL model will result with information that is inferred from the model, including statements not explicitly assigned to the `professor` individual. As an example consider that there is an inverse property `taughtBy` that assigns `ECE123` to have been taught by `professor`. This statement is enough for an OWL model to derive the inverse relation, without being directly assigned to the `professor` individial.

As Antoniou and van Harmelen (2004) pointed out "On the huge and only partially knowably World Wide Web, the open world assumption is the correct assumption... Nevertheless, a closed world assumption, (i.e. a statement is true when its negation cannot be proven) is also useful in certain applications" [8].

Unique names assumption: Typical database applications assume that individuals with different names are indeed different individuals [8]. This is also the default behavior in OO classes, unless implemented differently. In the contrary, OWL does not make such an assumption, rather it is controlled through specific constructs (`owl:sameAs`, `owl:sameAs`, `owl:differentFrom`).

[1] http://protege.stanford.edu

Inheritance issues: Semantic models in OWL/RDF allow multiple class inheritance, i.e. a class may have more than one parents. On the contrary, most programming languages allow a class to have only a single parent. However, multiple inheritance can be implemented in OO languages through the use of interfaces. Even recursive inheritance can be treated with the use of interfaces in OO languages. What is really unknown to OO languages is property inheritance. Properties in OO languages are second order entities that are defined through classes. For example, the member `name` of an `Employee` class is different from the member `name` of a `Person` class. Having said this, one realizes that property inheritance as defined in OWL is not supported by OO languages. For example, it can not be declared that the property `hasMother` is a specialization of `hasParent`. There are ways to implement such a feature, and overcome this issue, as we will discuss it in Section 3.3. In the relational paradigm there is no real construct for declaring formally inheritance of relationships.

Ownership of properties: Properties in semantic models are defined independently of classes. On the contrary, properties (or fields) in object models are defined through classes; they are members of classes. There is a direct ownership of properties in object models. As pointed out also in [6], in semantic models the statement `P rdf:domain D` does not mean that all members of `D` must have a property `P`, but that if a property `P` occurs for a resource `R`, `R` must be of type `D`.

Inversibility of properties Another issue related to the properties is the notion of an inverse property: Assume a model where a professor `teaches` some courses and each course `is taught` by some professors. In an OWL model, this is defined as an inverse relationship. In OO languages inversibility is required to be treated in code. In an OWL ABox is enough to state only an one-direction statement, and the OWL model infers the inverse direction relation. As an example, consider the following statement:

```
<owl:ObjectProperty rdf:about="#teaches">
    <owl:inverseOf rdf:resourse="#isTaught">
</owl:ObjectProperty>
```

An OWL reasoner will infer that the course `ECE123 isTaught` by `JohnDoe` only with the following statement:

```
<owl:Professor rdf:id="JohnDoe">
    <teaches rdf:resourse="#ECE123">
</owl:ObjectProperty>
```

The inverse statement is redundant. On the contrary, in Java we will need an implementation that manages inverse properties. According to the previous example, the `Professor` class needs the method:

```
public void addTeaches(ICourse course){
this.courses.add(course);
course.addIsTaught(this);
}
```

In relational models a many-to-many property is defined through an intermediate table, which doesn't really have a clear owner, so it fits better with the OWL notion of inversibility. In the above example, we can define two relations `Professor(id,...)` and `Course(id,...)` for the main entities and an intermediate relation `Teaches(professor,course)` with external keys to the main entity tables. This way, both ends of the property have access to it. Though this reads as, non-inverse properties cannot be enforced in a relational database, as there is no way to avoid traversing through the relations in both directions.

3 From semantic models to objects with relational persistence

Taking under account the differences discussed in the previous section, here we propose a transliteration of semantic models to object models with relational persistence. Our goal is to enable the software engineering process that aims to develop enterprise applications starting from a formal domain specification expressed in OWL (in the sublanguages of DL or Lite). This allows the logical verification of class hierarchy at design phase and the detection of inconsistencies through logical reasoning. It also enables the adoption of existing semantic models for domain specification.

3.1 Related work

Several toolkits are available for translating OWL/RDF structures into object models for supporting coding of semantic-rich applications. This is useful as the native programmatic interfaces for OWL/RDF, as OWL API[2], Jena[3], Protégé -OWL, seem arcane for most object-oriented programmers [9]. Undoubtedly, runtime access to ontologies has advantages (related to the execution 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 [10]. One of the first code generators from semantic models was the Protégé Bean Generator [11], which transforms conventional frame-based Protégé ontologies into Java source code for developing JADE agents [12]. Recent versions of Protégé -OWL incorporated code generation plugins that export Java source code following the conventions of the Eclipse Modelling Framework (EMF) cf. [13], or the JavaBeans. The Ontology Creator module in HarmonIA [14] adopts a more sophisticated approach that deals with multiple inheritance. Also, RDFReactor [15] is a toolkit for dynamically accessing an RDF model through domain-centric methods (getters and setters). ActiveRDF [6]is another dynamic object-oriented API for managing RDF data from Ruby programs.

The above tools provide with typical OWL/RDF storage options that rely on triplestores. Though triplestores are optimal for short statements, they are

[2] http://owlapi.sf.net
[3] http://jena.sf.net

quite slow compared to Relational Databases, when it comes to more complex queries against storages of realistic volume [16]. Also, relational databases is the industrial standard for data storage, and software engineers are much more comfortable with them.

Motivated from the above, we describe below how we can derive an object model with relational persistence from a semantic model expressed in idiomatic OWL, since only a subset of OWL/RDF constructs can be transliterated in a domain object model due to the impedance mismatch. Starting from a semantic model, defined in OWL, we transliterate it in an domain object model with the appropriate object-relational mappings that support persistence in a relational database. The key constructs of the three paradigms are aligned as shown in Table 1.

Note that individuals in OWL do not need to be directly typed, rather they are facts classified to classes, whose restrictions satisfy. This can be the case for individuals crawled on the Semantic Web. However, from an API generated from an OWL one would expect to accommodate only consistent individuals. Based on this requirement, we can consider objects as consistent individuals of the semantic model at hand.

Table 1. Model alignement

Semantic model	**Object model**	**Relational model**
class	class/interface	entity
property	field	attribute or associative entity
individual	object	tuple

3.2 Mapping OWL Classes

OWL classes are organized in a hierarchy using the `subClassOf` construct. Also, in OWL, there is a universal superclass for all classes (`owl:Thing`), and each individual is identified by a URI. Based on these remarks, we introduce in the domain object model, a superclass called `Thing`, that all generated classes inherit from. The `Thing` class has two fields, an `id` and a `URI`. The `id` is required for the persistent storage in a relational database, and in principle the `URI` is redundant, as each object in the generated object model can be uniquely identified through its id. The URI of an individual can be constructed by the URI of the database and the id as `jbdc:mysql://someserver/Database#12`, but for convenience we introduce the `URI` field. Also, the `Thing` class implementation provides with the appropriate functionality for the proper identification and equality of Things.

A class in OWL defines a group of individuals that belong together because they share some properties [17]. As such, it aligns better with an interface class that exposes these properties. Also, interface classes in object oriented languages support for multiple inheritance (diamond inheritance), in contrast with object

classes. Therefore, OWL classes are mapped into interfaces of the domain object model, and each one extends the `IThing` interface, that corresponds to the `Thing` class.

Classes in OWL/Lite are defined using `owl:Class` statements, that identify the URI of the class. Each one of them can be directly mapped to an interface in the domain object model, together with a class implementation. OWL/Lite also supports for logical intersections of classes or restrictions (`intersectionOf`), which are equivalent to a specification relationship, i.e can be implemented in the object model through interface class inheritance. Finally, OWL/DL supports for equivalent classes (`equivalentClass`), which can be read as the implementation of both interfaces.

OWL/DL, on top of the above, supports for enumerated classes. This resembles to enumerations in object oriented programming, though only enumerations of literals support relational persistence. Also, OWL/DL sublanguage provides with more boolean operations for classes (`unionOf`, `complementOf`, `disjointWith`). The union of two classes is equivalent to a common generalization relationship, therefore can be mapped through the inheritance of multiple interfaces. On the contrary, complement and disjoint relations are default behavior in object programming. The mapping from OWL Classes to object interfaces imposes a disjoint relationship, not always present in OWL. The mismatch between the semantic and object paradigms becomes apparent in this case. Despite those, the advantage for the programmer is that the domain model expressed in OWL can be logically verified through a reasoner, and the inferred class hierarchy can be used for deriving the domain object model.

The relational back-end for storing the content of the domain object model can be realized following the table per subclass pattern presented in [5] and implemented by most object-relational mapping toolboxes. The following table summarizes the mapping of OWL Classes to a domain object model with relational persistence.

Table 2. Summary of OWL Classes mapping with segments of Java code, and the corresponding tables for relational persistence.

OWL	OO	DB
`owl:Thing`	`Thing` `IThing`	`Thing(`*id*`,URI)`
owl:Class `A`	`A extends Thing implements IThing` `IA extends IThing`	`A(`*id*`,...)`

`A` equivalentClass `B`	`A extends Thing implements IB,IA,IThing`
	`IA extends IThing`
	`B extends Thing implements IB,IA,IThing`
	`IB extends IThing`
		`A(`*id*`,...)`
		`B(`*id*`,...)`

`A` intersectionOf `B` and `C`	`A extends Thing implements IB,IC,IA,IThing` `IA extends IB,IC, IThing`	`A(`*id*`,...)`

`A` unionOf `B` and `C`	`B extends Thing implements IB,IA,IThing`
	`IB extends IA,IThing`
	`C extends Thing implements IC,IA,IThing`
	`IC extends IA,IThing`
		`B(`*id*`,...)`
		`C(`*id*`,...)`

3.3 Mapping properties and constraints

OWL properties are mapped into members or fields of classes and accessor and mutator methods in the interfaces. 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 both cases, their transliteration in the domain object model depends on the cardinality of the property, which in OWL can de defined universally on the property, or as a constraint for the property in the class definition. Singular cardinality properties are mapped to references of single objects, while multiple cardinality ones are mapped as sets of objects.

OWL **literal properties** are defined using the XML Schema datatypes, which support most object languages and databases. For example, let a singular property `price` of a class `Garment` to have a float range. This corresponds to a field `Float price`, accompanied by the accessor and mutator methods `Float getPrice()` and `void setPrice(Float f)`. In the case the price property has a maximum cardinality greater than one, it can be mapped to a field `Set<Float> price`. Using object-relational mappings, the single cardinality literal properties are added as attributes in the `Garment` table, while for multiple cardinality literal properties, an associative table `GarmentPrices(id, price) key:id=Item.id`.

For **object properties** the principles for adding them in the object model are the similar: they map to members and methods of the class and the interface. The only shortcoming is that the range of an object property in OWL can be either a named classes, or an axiom. In case of a range set to a named classes, the corresponding field of the domain class and the signature of the methods refer to the interface of the class in range. For example, consider an OWL class `CatOwner` that through the object property `isOwnerOf` is associated to a set of `Cat` classes. The generated interface `ICatOwner` will include the methods `Set<Cat> getIsOwnerOf()` and `void setIsOwnerOf(Set<Cat> set)`. In case an object property range is de fined though an axiom, the corresponding interface of the axiom is used. Anonymous axioms can be given arbitrary names, for example using the OWL Manchester syntax. (i.e. from `Apples or Pears` syntax we can name an interface as `IApplesOrPears`). Finally, the object model can accommodate cardinality constraints of certain ranges by adding checks in the accessor and mutator methods, to ensure consistency. Also, transitive and symmetric properties can be implemented in code as well. What remains complex to be implemented in the object model is the notion of functionality, which corresponds by definition to a unique attribute relation in a database. We will discuss this below, where we present the relational mappings of properties.

Aligning OWL object properties to an object model with relational mappings is a complex task, as the type of the underlying relational schema mapping depends on the inversibility of the property and the cardinality constraints on both sides. 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, standard object-

Table 3. Object-relational mapping of an OWL object properties is defined by cardinality and inversibility

inversibility \ cardinality	singular	multiple
not inverse	1-1 unidirectional	N-M unidirectional
inverse of a singular	1-1 bidirectional	N-1 bidirectional
inverse of a multiple	1-N bidirectional	N-M bidirectional

relational mapping tools provide support for directions in relationships, which means that an inverse OWL property is mapped in both sides of the relationship in the object model, but it could refer only to one attribute or associative table in the relational incarnation. We have discussed in detail these mappings in [3], and we present an overview of all cases in Table 3. Note that the relational back-end provides with a native implementation of functional OWL properties, whose attributes can be declared as unique.

Finally, mapping `rdfs:subClassOf` is not straightforward issue, as OWL treats properties as first order entities, which exist independently from classes and form hierarchies. We pointed out also in Section 2 that domain object models do not have similar functionality. For imitating the behavior of property inheritance a possible solution is to have child properties calling the father properties. For example, a mutator `setMother(Woman w)` should also invoke the mutator of the superproperty `setParent(Person w)`. However, such a solution is expensive due to the duplication of data, and also conflicts with the relational database principles.

3.4 Implementation

We implemented the methodology presented above as a plugin for Protégé -OWL (v.3.4), named SeRiDA from Semantic-Rich Development Architecture and it will become available as a open-source project at: `http://serida.sf.net`. The export plugin traverses in a single pass through all concepts in an ontology and generates the corresponding JavaBeans source code and Hibernate mappings[4], as both of them are widely used in enterprise application development. The current version doesn't support property inheritance, transitive and symmetric relations. We intend to develop a plugin for Protege 4 as well.

4 Evaluation

4.1 Evaluation with LUBM benchmark

In order to evaluate the performance of our framework in querying large repositories, we selected the Lehigh University Benchmark (LUBM) [18]. LUBM provides with an evaluation framework for semantic web repositories. It includes

[4] http://www.hibernate.org

an OWL ontology that refers to the university domain[5]; an instance generator, which provides with synthetically generated data; a set of extensional queries expressed in SPARQL; and a set of performance metrics.

Using the developed Protégé plugin for exporting JavaBeans and Hibernate Mappings, we exported the LUBM ontology into the corresponding object model with relational persistence. In order to simplify the generated object model and to be able to store all synthetic data generated by LUBM framework, the LUBM ontology was extended as follows: (a) domain and range restrictions were added in some properties, so that they will not appear in all generated classes. (b) `TeachingAssistant` class was declared as a subclass of a `GraduateStudent`.

In order to generate the synthetic data, we used the instance generator provided with the LUBM framework. It gets two parameters, one is the number of universities to generate (u) and the second is the seed for the pseudo-random generator (s). Synthetic datasets are identified in the LUBM(u, s) notation. The instance generator outputs an RDF document. We extended this architecture for generating Java code that uses the generated JavaBeans to persistently store data.

Then, we expressed all fourteen LUBM queries in the Hibernate Querying Language (HQL), and used Hibernate to execute all of them against the LUBM(1,0) data set. The first query written in SPARQL and HQL is shown below:

```
# Query1 in SPARQL
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/zhp2/2004/0401/univ-bench.owl#>
SELECT * WHERE
{
?x rdf:type ub:GraduateStudent .
?x ub:takesCourse <http://www.Department0.University0.edu/GraduateCourse0> .
}

# Query1 in HQL
select o
from
edu.lehigh.zhp2.s2004.s0401.univ_bench.IGraduateStudent as o
where
o.TakesCourse.URI='http://www.Department0.University0.edu/GraduateCourse0'
```

Note that the result of the query in SPARQL is a set of URIs, while the result in HQL is a set of objects implementing the `IGraduateStudent` interface.

As Hibernate enables polymorphic queries, the selected object `o` can be of any type that implements the `IGraduateStudent` interface. HQL can resolve the corresponding tables, access them in the database and return the results. This also demonstrates how the inferred hierarchy can be used by simply adding interfaces in the generated Java code. New classes can be inserted in the model

[5] http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl

Table 4. Summary of the querying results against LUBM(1,0) data using HQL.

Query	**Time response**		**Answers**	**Measures**	
	mean	(std)	act/exp	F	P
Q1	126.42	(4.33)	4/4	1	0.987
Q2	27.42	(1.44)	0/0	1	0.992
Q3	389.30	(4.16)	6/6	1	0.955
Q4	17.03	(3.22)	34/34	1	0.993
Q5	381.41	(6.06)	678/719	0.94	0.957
Q6	709.70	(47.98)	7790/7790	1	0.810
Q7	1,411.48	(36.14)	67/67	1	0.113
Q8	933.55	(49.51)	7790/7790	1	0.582
Q9	2,234.09	(41.53)	208/208	1	0.002
Q10	1,257.46	(40.16)	4/4	1	0.216
Q11	483.57	(40.67)	0/224	0	0.930
Q12	504.23	(39.25)	0/15	0	0.923
Q13	494.14	(37.60)	0/1	0	0.926
Q14	539.37	(39.59)	5916/5916	1	0.909

Note: Time response is in msec per query. F and P are performance measures of the LUBM framework described in the text.

without changing the relational back-end: only the generated JavaBeans need to be extended with new interfaces.

Then, we executed all LUBM queries 1000 times on an iMac with a 3.06 GHz Intel Core 2 Duo processor with 4GB of RAM, using Java 1.5 with 1GB of maximum heap size and an underlying MySQL Server (ver 5.1) running on the same machine. Actual results and the average response time for all queries is shown in the following Table 4.1.

The results above demonstrate that using HQL support for polymorphic queries bears with correct and sound answers for queries Q1-Q4, Q6-Q10 and Q14, that require generalization in the object class hierarchy. This is achieved by using Java reflection and implementing the corresponding interfaces of all inferred parent classes. Implicit subclassing included in queries Q6-Q10 has been treated soundly, as the inferred hierarchy has been used for the generated JavaBeans.

Then, Q11 requires traversing through `subOrganizarionOf` which is a transitive property. Transitivity is not supported in HQL, therefore the corresponding HQL query returns no results. Queries Q5, Q12 and Q13 require traversing through sub-properties, which is again not supported in HQL. In Q5 sound but not complete results have been retrieved, while Q11-Q13 didn't result any. We proposed a solution for this problem of sub-properties in Section 3.3.

The average time response remained below 1ms in most of the queries, with the most expensive one being the Q9 that is characterized by the most classes and properties in the query set and there is a triangular pattern of relationships.

Finally, we calculated some metrics proposed by the LUBM framework for measuring querying performance. The F measure computes the tradeoff between

answer completeness and soundness [18], as $F_q = \frac{(\beta^2+1)C_q S_q}{\beta^2 C_q + S_q}$ and the query performance metric $P_q = \frac{1}{1+e^{a T_q/N-b}}$, where for query q C_q is completeness, S_q is soundness and T_q is the time response. Parameters were set to the values indicated in [18], i.e. $a = 500, b = 5, \beta = 1$ and $N = 100\,000$, in order to respond with a $P(q) = 0.5$ when a query is responded in 1ms. Table 4.1 presents these measures for all queries. In 10 out of 14 queries F measure is one, as sound and complete results have been retrieved, while the P measure tends to 1 for those queries that have been answered faster.

4.2 Cross-evaluation with semantic repositories

Another incentive to adopt an object-oriented model for managing data with relational persistence can be its performance with respect to other repositories, especially the semantic web ones. A recent comparison between semantic and relational storages [16] demonstrated that there is a great difference with respect to their query-answering performance. Relational storage is much faster. The SeRiDA framework stands somewhere in between of the two choices, as it does not offer the full functionality of a semantic data storage, but certainly offers more than a conventional RDBMS. For example the support for polymorphism and polymorphic queries can be considered as a classification service.

For getting a rough evaluation of the performance of SeRiDA against semantic repositories, we compared its performance against the Jena framework. As we have in mind to evaluate the scaling of the approaches with much larger data-sets, we installed Jena/SDB[6], which provides with RDF storage using SQL databases, including transactional operations. Jena/SDB was installed on the same machine with SeRiDA, using the same Java and MySQL configurations. Using LUBM benchmark we generated the LUBM(1,0) dataset in RDF to be loaded on Jena/SDB. However, as Jena/SDB is intended for RDF storage, we preprocessed the synthetic data by applying the Jena reasoner. The extended datasets that include inferred data have been loaded in Jena/SDB. Then, we executed each LUBM benchmark queries 1000 times and recorded the average time response. A comparison of the SeRiDA and Jena performance are shown in Table 5 for selected queries, which does not require any inference, or require only class generalization inference.

As a metric of comparison of the performance of the two systems we selected the LUBM composite metric $CM = \sum_q w_q \frac{(\alpha^2+1)P_q F_q}{\alpha^2 P_q + F_q}$, where α determines the relative weight between F and P. We set $\alpha = 1$, as suggested in [18].

Table 5 suggests that SeRiDA response has been well below 1ms per query, for most of the queries. Also SeRiDA is faster than Jena/SDB, except for Q6 and Q14. Both Q6 and Q14 of the are simple queries with low selectivity. Q6 asks for all instances of type `Student` and Q14 for all instances of `UndergraduateStudent`. The former is more complex, as it assumes the subclassing relationship. This remark draws us to the conclusion, that the underlying ORM approach has an

[6] http://jena.hpl.hp.com/wiki/SDB

Table 5. Comparison of query response time between Jena/SDB and SeRiDA: The average query response time (ms) is shown for the selected queries, along with F and P measures. The composite metric for all queries has been calculated as well.

	SeRiDA				**Jena/SDB**			
Query	Time response		Measures		Time response		Measures	
	mean	(std)	F	P	mean	(std)	F	P
Q1	**126.42**	(4.33)	1.00	0.987	920.29	(33.96)	1.00	0.598
Q3	**389.30**	(4.16)	1.00	0.955	2,655.20	(57.75)	1.00	0.000
Q4	**17.03**	(3.22)	1.00	0.993	322.82	(3.17)	1.00	0.967
Q5	**381.41**	(6.06)	0.94	0.957	3,507.32	(9.17)	1.00	0.000
Q6	709.70	(47.98)	1.00	0.810	**126.46**	(2.55)	1.00	0.987
Q10	**1,257.46**	(40.16)	1.00	0.216	3,312.75	(9.21)	1.00	0.000
Q14	539.37	(39.59)	1.00	0.909	**96.29**	(1.56)	1.00	0.989
CM Index	3.139				1.939			

overhead that becomes apparent in simple queries, while it becomes less important for complex ones. It also verifies the fact that triplestores are very efficient for simple queries.

5 Discussion and future work

This paper presented our work in progress for using idiomatic OWL/RDF models as a domain specification language for object modelling with relational persistence. We presented a methodology that transliterates OWL models to object models with relational persistence, and we coded a Protégé plugin that implements it. Finally we report our evaluation of the performance of the method against the LUBM benchmark.

We realized that the transliteration from OWL models to domain object models, though not complete due to the semantic-object-relational impedance mismatch, it still provides with a powerful tool for the programmer: a systematic way to model a domain using Description Logics, that can verify the model consistency at early design stage through reasoning. Also, the transliteration may be proven useful for deploying end-user applications that rely on object-oriented programming and relational databases, but originate from existing OWL ontologies. In this way, the developments of the Semantic Web community in terms of OWL/RDF domain ontologies can indirectly contribute to engineering software applications.

Finally, the presented method appears to significantly improve the query response times, which means that have a potential as a semantic storage platform as well.

Future work will focus on investigating further the performance of SeRiDA in query response, involving larger datasets from the LUBM framework, in order to evaluate the scaling performance. Also, the implementation of the property sub classing needs further investigation, especially in terms of performance. Finally,

we intend to add an inspection facility that will identify axioms/patterns in an ontology that do not allow its transliteration into a domain object model. Adding missing axioms in the OWL, so that the translation becomes more explicit at a logical level may be useful as well.

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 Specification version 1.0, OMG (2009)
3. Athanasiadis, I.N., Villa, F., Rizzoli, A.E.: Enabling knowledge-based software engineering through semantic-object-relational mappings. In Kendall, E.F., et al., eds.: Proceedings of the 3rd International Workshop on Semantic Web Enabled Software Engineering, 4th European Semantic Web Conference, Innsbruck, Austria, KnowledgeWeb (2007) 16–30.
4. Athanasiadis, I.N., Villa, F., Rizzoli, A.E.: Ontologies, JavaBeans and Relational Databases for enabling semantic programming. In: Proc. of the 31th IEEE Annual International Computer Software and Applications Conference (COMPSAC). Volume 2., Beijing, China, IEEE (2007) 341–346 First IEEE International Workshop on Development and Application of Knowledge Based Software Engineering Tools.
5. Ambler, S.W.: Building object applications that work. Cambridge University Press (1998)
6. Oren, E., Heitmann, B., Decker, S.: ActiveRDF: Embedding Semantic Web data into object-oriented languages. Web Semantics: Science, Services and Agents on the World Wide Web **6** (2008) 191202
7. Motik, B., Horrocks, I., Sattler, U.: Bridging the gap between OWL and relational databases. Web Semantics: Science, Services and Agents on the World Wide Web **7** (2009) 74–89
8. Antoniou, G., van Harmelen, F.: A Semantic Web Primer. MIT Press (2004)
9. Knublauch, H., Horridge, M., Musen, M., Rector, A., Stevens, R., Drummond, N., Lord1, P., Noy, N.F., Seidenberg, J., Wang, H.: The Protégé OWL experience. In: Workshop on OWL: Experiences and Directions. (2005)
10. 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)
11. van Aart, C., Pels, R., Caire, G., Bergenti, F.: Creating and Using Ontologies in Agent Communication. In Cranefield, S., Finin, T., Willmott, S., eds.: Ontologies in Agent Systems, 1st International Joint Conference on Autonomous Agents and Multi-Agent Systems. Volume 66 of CEUR Workshop Proceedings., Bologna, Italy (2002)
12. Bellifemine, F., Caire, G., Poggi, A., Rimassa, G.: JADE-A white paper. EXP in search of innovation **3** (2003) 6–19
13. 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)

14. 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 (2004)
15. Völkel, M.: RDFReactor - From Ontologies to Programmatic Data Access. In: International Semantic Web Conference ISWC-2005. (2005)
16. Bizer, C., Schultz, A.: The Berlin SPARQL Benchmark. International Journal on Semantic Web & Information Systems **5** (2009) 1–24
17. Guinness, D.L.M., van Harmelen, F., et al.: OWL Web Ontology Language overview. W3C Recommendation, W3C (2004) www.w3.org/TR/owl-features/.
18. Guo, Y., Pan, Z., Heflin, J.: LUBM: A benchmark for OWL knowledge base systems. J. Web Sem. **3** (2005) 158–182