In **Proc. of the 31th IEEE Annual International Computer Software and Applications Conference (COMPSAC)**, Beijing, China, IEEE, July 2007, pp.341-346.

Ontologies, JavaBeans and Relational Databases for enabling semantic programming

Ioannis N. Athanasiadis
IDSIA/USI-SUPSI
Lugano, Switzerland
ioannis@idsia.ch

Ferdinando Villa
Univ. of Vermont
Burlington, VT, USA
fvilla@uvm.edu

Andrea-Emilio Rizzoli
IDSIA/USI-SUPSI
Lugano, Switzerland
andrea@idsia.ch

Abstract

Knowledge-based software engineering enables a programmer to integrate rich semantics in the software development process. In this work, we show how an OWL/RDF knowledge base can be integrated with conventional domain-centric data models (Enterprise Java Beans) and object-relational mapping toolkits (Hibernate). We present a pathway for the software developer to generate enterprise Java beans source code and hibernate object-relational mappings starting from a domain ontology. This way, a semantic-rich enterprise development environment is specified that combines the benefits of using ontologies with software development standards.

1 Introduction

Building semantic-rich applications requires the synthesis of traditional AI concepts with every-day software engineering practice. Domain-specific conceptualizations are increasingly specified as formal ontologies, as part of ongoing efforts for enabling the semantic web. However, 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. One of the major issues raised is that using a semantic model to its full extent results in complex software interfaces, with several degrees of freedom. For example, existing programming toolkits handle OWL individuals in a triple-store approach. Though powerful, such a practice is very unfamiliar to the conventional programmer, who feels comfortable with standard domain models for building domain applications. Similarly, common strategies to archive individuals in RDF triple-stores or in uniform "subject-predicate-object" relational database tables hardly fit the notion of normalized relational databases.

In this paper we present how OWL semantic models can be used to specify and build client-server enterprise applications. We present a three-layer framework that translates ontology constructs into Enterprise Java Beans, enabling easy software coding, and connects them to relational database persistence storage through the generation of Hibernate object-relational mappings. In the context of knowledge-based software engineering, the framework presented in this paper demonstrates how semantic models can be coupled with standard programming practices for building semantic-rich applications.

The rest of the paper is structured as follows; Section 2 summarizes related work on both linking ontologies and databases, and generating programmatic interfaces from ontologies. Based on these findings, we present in section 3 an abstract architecture for semantic programming that combines both generated programmatic interfaces with relational back-end storage of individuals. Section 4 shows how a domain model specified using an ontology can be translated to both enterprise Java Beans and relational storage. The main findings of this paper are summarized in section 5.

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) populating ontologies with instances initially stored in relational databases. Various software tools and libraries exist to enable (1), e.g. Jena [13], Protégé [8] and KAON [11]. 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 (i.e. classes) 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 [20]. 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 [15]. Therefore, alternative kinds of OWL instances storage that are optimized for indexing and retrieval are required. Roldan Garcia and Aldana-Montes [15] 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 [3, 4] 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 [6, 17] 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 heterogenous relational databases using semantic web technologies [5]. 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 of semantic applications, apart the ontology development toolkits mentioned above. One of the first translators was the Protégé Bean Generator [18], which transforms conventional frame-based Protégé ontologies into Java source code for developing JADE agents [2]. 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. [16]. The RDFReactor approach [19] 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. [10], that deals with issues as multiple inheritance. However, both of them still store the generated instances as triple-stores.

3 A semantic programming architecture

Building upon these contributions, we propose a semantic-rich development architecture that combines conventional enterprise development software practices as Enterprise Java Beans [7] and Hibernate object-relational mapping [9] for persisting the content of the generated java classes.

To speed up end-user application coding in the context of knowledge-based software engineering, semantic modelling practices must interface with software developer needs. In his vision of ontology-driven software development, Knublauch [12] recommends that 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. Going a step further, Knublauch envisions a programming practice that instead of relying on UML defines the domain model in OWL.

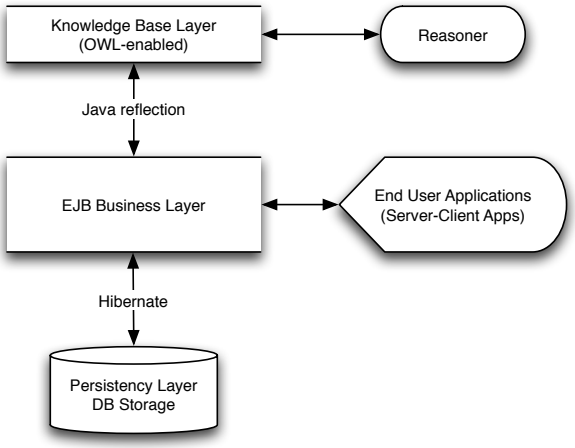


Figure 1. The semantic-rich programming platform architecture

A semantic-rich programming platform needs to synthesize the appropriate 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 presented in Fig. 1.There are two modes of operation in such an environment, explained below.

On one hand, starting from the higher level of semantic modeling, we can generate code automatically, using conventions presented in the following section. Assume that an ontology is given, that specifies the conceptualization of a domain. Part of this semantic model specifies the concepts involved, and can be translated into data structures and entities specifications, while a second part defines the logics that pertains to these concepts. Based on the data structure specifications of the domain ontology we can generate both the programming interface and a normalized persistence storage in a database. This part is detailed in section 4. Having generated both the programming interface and the relational schema, the platform enables a semantic-rich framework for software development.

On the other hand, by keeping track of the original OWL-

Classes used for the generated programming interface (e.g. through Java reflection), we can connect the generated java objects back to the semantic layer (i.e. in the knowledge base) and apply a reasoner on them. So for example, we can classify an object of a generated Java class according to ontology-defined classifications that were not present in the knowledge base when source code was generated. In this way, we consider the formal specifications of domain knowledge expressed in a semantic model using description logics, as an upper layer for storing part of the business intelligence that can be updated at runtime, without affecting the conventional APIs for coding and application development.

4 From OWL models to coding interfaces and relational database schemes

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 [10], 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.

4.1 From OWL Classes to Java Classes and Entities

In a relational representation of an ontology, each OWL Class typically represents one entity. However, an OWL Class pertains to both the data structure and the Description Logics of entities. It is quite common to have OWL classes that do not assign additional data properties to their father 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 patterns are consider part of the "business" logic of the ontology, and do not contribute axioms of relevance to the relational model, that aims to persist the ontology-defined data structures.

We consider eligible for persistent storage only those non-anonymous OWL classes that contribute with additional attribute specifications in the class inheritance. Each of those classes is considered to represent an entity and is 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`. The table `csPerson` will be extended with attributes and relations that derive from the OWL Properties of Class `cs:Person`.

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.

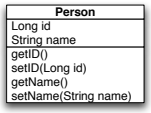
4.2 Literal properties

Literal properties simply specify data type attributes of an entity. 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.

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:



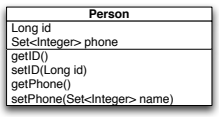
I-b: Multiple cardinality literal property A literal property of multiple cardinality identifies a multi-valued attribute of an entity, dependent only upon the primary key. Multi-valued attributes in normalized database systems are implemented as associate entities, through an one-to-many relationship. In the `cs:Person` example, lets include a literal property `cs:phone` as:

```
<owl:DatatypeProperty rdf:ID="phone">
  <rdfs:domain rdf:resource="#Person"/>
  <rdfs:range rdf:resource="&xsd;int"/>
</owl:DatatypeProperty>
```

without any additional cardinality constraints. 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 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. 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.

4.3.1 Non-inverse properties

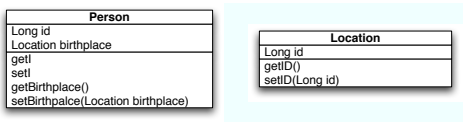
II-a Non-inverse functional (singular cardinality) object property An non-inverse functional of singular cardinality object property can be translated as one-to-one unidirectional relationship, which is added as an attribute in the owning entity. For example, let `cs:Person` have a functional property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



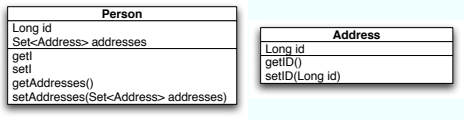
II-b Non-inverse object property In the general case, an non-inverse multi-cardinality object property defines a many-to-many unidirectional relationship between two entities. For example, an object property `hasAddress` may associate each `cs:Person` class to several `cs:Address` classes (we assume 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:



4.3.2 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>
```

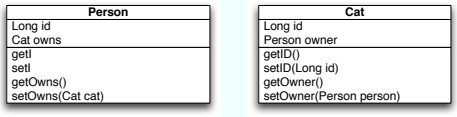
In this example,the `owl:FunctionalProperty` construct is used for OWL code simplicity, instead of a cardinality constraint. The content of both `cs:owns` and `cs:hasOwner` properties can be stored simply as an extra attribute of any of the two entities, as:

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

Or alternatively can be realized as an intermediate 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 is the DB schema, the `Person` JavaBean will have an attribute 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 In this way we specify a bi-directional one-to-many relationship, that can be implemented with an intermediate 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:

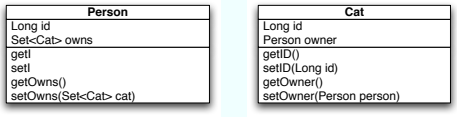
```
<owl:ObjectProperty rdf:ID="owns">
  <rdfs:domain rdf:resource="#Person"/>
  <rdfs:range rdf:resource="#Cat"/>
  <owl:inverseOf rdf:resource="#hasOwner"/>
</owl:ObjectProperty>

<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 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 JavaBean level, the `Person` class has an attribute `owns` that refers to a set of `Cat` objects.



II-e Object property inverse of a object property In the generic case that there are not any cardinality restrictions, 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:

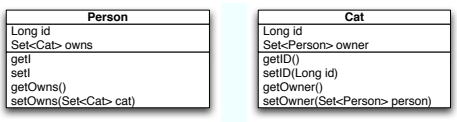
```
<owl:ObjectProperty rdf:ID="owns">
   <rdfs:domain rdf:resource="#Person"/>
   <rdfs:range rdf:resource="#Cat"/>
   <owl:inverseOf rdf:resource="#hasOwner"/>
</owl:ObjectProperty>

<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 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 attributes `Set<Cat>` and `Set<Cat>` respectively.



5 Discussion

We have implemented a translator that generates programming interfaces as Enterprise Java Beans and Hibernate object-relational mappings from OWL ontologies. The translator is a plugin for the integrated knowledge management toolkit ThinkLab[1] and is available online[2]. The presented semantic-rich programming framework is currently used in the Seamless-IP project for linking agronomic models and environmental data across scales and disciplines, using ontologies. In the current implementation multiple inheritance in OWL is implemented using interface implementations in object-oriented modelling, while properties inheritance is not supported. A demonstration applied in modelling farming systems and management alternatives of a farm household is presented in [1].

Future efforts will concentrate on issues related to classifications, inheritance and polymorphism, while a persistence plugin for Protégé-OWL may be developed.

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 Framework Programme Integrated Project SEAMLESS (contract no. SUSTDEV-10036), and the National Science Foundation, award DBI 0640837 for "Project ARIES".

References

[1] I. N. Athanasiadis, S. Janssen, D. Huber, A. E. Rizzoli, and M. van Ittersum. Semantic modelling in farming systems research: The case of the agricultural management definition module. In J. Marx Gómez, M. Sonnenschein, M. Müller, H. Welsch, and C. Rautenstrauch, editors, *Information Technology in Environmental Engineering (ITEE 2007)*, pages 417–432, Oldenburg, Germany, March 2007. ISCS-NAISO, Springer-Verlag.

[2] F. Bellifemine, G. Caire, A. Poggi, and G. Rimassa. JADE-A white paper. *EXP in search of innovation*, 3(3):6–19, September 2003.

[3] C. Bizer. D2R map: A database to RDF mapping language. In *Twelfth International World Wide Web Conference (WWW2003)*, Budapest, Hungary, 2003.

[4] C. Bizer and R. Cyganiak. D2R-server:publishing relational databases on the web as SPARQL-endpoints. In *15th International World Wide Web Conference (WWW2006)*, Edinburgh, UK, 2006.

[5] H. Chen, Y. Wang, H. Wang, Y. Mao, J. Tang, C. Zhou, A. Yin, and Z. Wu. Towards a semantic web of relational databases: A practical semantic toolkit and an in-use case from traditional chinese medicine. In I. F. Cruz, S. Decker, D. Allemang, C. Preist, D. Schwabe, P. Mika, M. Uschold, and L. Aroyo, editors, *5th International Semantic Web Conference*, volume 4273 of *Lecture Notes in Computer Science*, pages 750–763. Springer, 2006.

[6] O. Curé and R. Squelbut. Semantic mapping to synchronize data and knowledge bases at the instance level. In *European Semantic Web Conference*, 2006.

[7] L. DeMichiel and M. Keith. Enterprise javabeans 3.0 specifications. Java Specification Request 220, 2006.

[8] M. Horridge, H. Knublauch, A. Rector, R. Stevens, and C. Wroe. 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, August 27 2004.

[9] JBoss. Hibernate reference documentation, User manual, available online: http//www.hibernate.org

[10] A. Kalyanpur, D. J. Pastor, S. Battle, and J. Padget. Automatic mapping of owl ontologies into java. In *16th Int'l Conference on Software Engineering and Knowledge Engineering*, Banff, Canada, June 2004.

[11] KAON. The Karlsruhe ontology and semantic web framework developer's guide for kaon 1.2.7. Technical report, University of Karlsruhe, Germany, 2004.

[12] H. Knublauch. Ramblings on agile methodologies and ontology-driven software development. In *Workshop on Semantic Web Enabled Software Engineering, International Semantic Web Conference*, Galway, Ireland, 2005.

[13] B. McBride. Jena: A semantic web toolkit. *IEEE Internet Computing*, 6(6):55–59, 2002.

[14] N. Moreno, I. Navas, and J. Aldana. Putting the semantic web to work with DB technology. *Bulletin of the IEEE Technical Committee on Data Engineering*, 26(4):49–54, 2003.

[15] M. d. M. Roldan Garcia and J. F. Aldana-Montes. A tool for storing owl using database technology. In B. C. Grau, I. Horrocks, B. Parsia, and P. Patel-Schneider, editors, *First Int'l Workshop on OWL Experiences and Directions*, Galway, Ireland, 2005.

[16] D. K. Sharma, T. M. Johnson, H. R. Solbrig, and C. G. Chute. 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, July 2005.

[17] R. Squelbut and O. Curé. Integrating data into an owl knowledge base via the dbom protege plug-in. In *8th Intl. Protégé Conference*, 2006.

[18] C. van Aart, R. Pels, G. Caire, and F. Bergenti. Creating and using ontologies in agent communication. In S. Cranefield, T. Finin, and S. Willmott, editors, *Ontologies in Agent Systems, 1st International Joint Conference on Autonomous Agents and Multi-Agent Systems*, volume 66 of *CEUR Workshop Proceedings*, Bologna, Italy, 2002.

[19] M. Völkel. RDFReactor - from ontologies to programmatic data access. In *International Semantic Web Conference ISWC-2005*, 2005.

[20] K. Wilkinson, C. Sayers, H. Kuno, and D. Reynolds. Efficient rdf storage and retrieval in Jena2. In *Proceedings of VLDB Workshop on Semantic Web and Databases*, pages 131–150, 2003.

[1] `http://www.integratedmodelling.org`

[2] `http://imt.svn.sourceforge.net/svnroot/imt/Thinklab/`