Semantic Annotation for Persistence

Stephen L. Reed

Texai.org
3008 Oak Crest Avenue
Austin, Texas USA 78704
stephenreed@yahoo.com

Abstract

An enhancement of the Java Persistence API is described in which source code annotations contain meaning-denoting statements. Java domain entities are persisted in a knowledge base that is implemented by a database, rather than persisted in a database directly. The knowledge base format facilitates information exchange via the Semantic Web by having a reusable commonsense ontology.

Introduction

Application developers who seek to benefit from information exchange via the Semantic Web have to develop a new ontology, or re-use an existing ontology that describes the meaning of the domain concepts and the meaning of their properties. Often, applications written in an object-orientated language, such as Java, store and retrieve database information as objects using an object/relational mapping (ORM) technique. The developer then has the burden of either hand-coding database access methods for object persistence, or specifying the object/relational mapping to a framework such as Hibernate (Bauer and King, 2007) in XML. The most recent releases of Java (beginning with version 1.5) greatly improve the ease of mapping program object instances to relational database tables and columns. Java 1.5 introduced source code annotations, including a set designed for object persistence, (Sun Microsystems, 2006 and Bauer and King, 2007).

Our research extends the notion of object persistence annotation to include semantic attributes. We reuse the ontology of OpenCyc (Matuszek et. al., 2006), a large commonsense knowledge base that is freely available. We stored the ontology of OpenCyc, the content of the WordNet lexical database, the content of Wiktionary, and the content of the CMU Pronouncing Dictionary in a relational format that we have named the Texai Knowledge Base (KB). The annotation attribute values designate Texai KB concept classes, context, and properties that hold between persisted domain entities. This semantic information is required to utilize the Texai KB.

Copyright © 2007, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

The chief benefit of our technique is to move the application data from a semantically impoverished relational database into a semantically rich KB. In particular, interfacing to the Semantic Web is generally easier using a KB rather than a database. Additionally, the application benefits from the ability to reason about the domain objects within the KB by using class or property subsumption with respect to context. Henceforth we shall refer to a Java domain object instance as a ***domain entity***.

Brief Illustration

By way of illustration, consider a simple Java use case in the health science domain. Two of the domain entities are as follows: a bird flu epidemic and the location in which the epidemic occurred.

```
public class BirdFluEpidemic {
  ...
  private String location;
  ...
}
```

To publish this data on the Semantic Web, the developer must create an adapter that describes the semantic class of the epidemic and location domain entities and determine the appropriate semantic property that holds between pairs of these domain entities. With our technique, semantic annotations authored by the developer declare the required information in the Java source code *during the initial design* of the class.

```
@DomainEntity(type="TemporalStuffType",
subclassOf="Epidemic")
public class BirdFluEpidemic {
  ...
  @DomainProperty(
    name="epidemicExposureOccursAt")
  private GeographicRegion location;
  ...
}
```

The Java annotation **@DomainEntity** is defined by our

framework and, as used here, declares that BirdFluEpidemic is a class in the Texai KB, has TemporalStuffType as its type, and that it is a subclass of the class Epidemic. The BirdFluEpidemic class will be automatically defined in the KB using these attributes if not already present.

The annotation **@DomainProperty** declares that the KB property has the name epidemicExposureOccursAt. By default, the property domain, which is its first argument, must be an instance of the KB BirdFluEpidemic class. The second argument is the property range and it defaults to an instance of the KB GeographicRegion class. In this example, the first argument type constraint is implicitly given by the domain entity class, and the second argument type constraint is implicitly given by the field type declaration. Note that the location type has been changed to the semantically meaningful GeographicLocation type rather than the semantically opaque String type.

A hypothetical instance of the Java BirdFluEpidemic class having the identifier BirdFlu_1 that was published in London could easily be expressed in the Semantic Web notation RDF with the following statement. It is expressed as an RDF triple consisting of *rdf-subject, rdf-property, rdf-object*:

```
BirdFlu_1 exposureOccursAt CityOfLondonEngland
```

Java Object Persistence

Large scale applications are often developed in the Java programming language and may have large amounts of data stored in relational database tables. Because each individual program execution is transient, application data cannot usually be initialized from the program source code; rather, it must be saved in the database when created or updated, and then retrieved from the database. That data is used to construct Java objects, in the application as needed. These objects are defined as ***persistent***. In the past, developers had to manually specify the mapping between database fields, residing in named tables/columns, and Java object variables. Developers had to create code to load/store/update each Java object from the database with explicit SQL operations. Yoder et. al., (1998) describe a set of implementation patterns for Smalltalk, a classic object-orientated language that preceded and influenced Java. In recent years, declarative frameworks have become available to automatically perform the object-relational mapping, some going so far as to create the database schema given only the persistent Java class definitions (Frederickson, 2006). Presently, the most popular of these frameworks is Hibernate (Bauer and King, 2007) which includes an implementation of the new Java Persistence API. This API is also a key feature of the latest version of the Java 2 Enterprise Edition (J2EE) software, which greatly simplifies the development of persistent Java objects in large scale enterprise applications compared to the previous version that required implementing framework persistence interfaces.

Texai Knowledge Base

The Texai KB supports the usage of semantic annotation for persistence. It contains commonsense facts, formulas and rules that facilitate the development of intelligent applications (Reed, 2006). A wide variety of application domains can be uniformly represented in the KB using a comprehensive open-source ontology. Java language bindings and in particular, the use of specialized Java source code annotations make this KB interesting for Java application developers. Annotated Java domain entities can be persisted to and loaded from a set of propositions in the KB.

Although this method of persistence is slower than the conventional method of persisting Java objects to a single database row, for Semantic Web applications there are benefits in having the data stored in an RDF-compatible format. Chief among these benefits is the ability to easily export the data as RDF triples. Another benefit occurs when the developer begins with a set of RDF triples and desires to create a Java application that processes this data. The Texai KB enables a concise object modeling of the RDF data graph as semantically annotated classes that define the domain entities.

The Texai KB has, as its foundation, the ontological definitions imported from the OpenCyc KB, version 1.1. Cyc is described as "a large knowledge base containing a store of formalized background knowledge suitable for a variety of reasoning tasks in a variety of domains" in Matuszek et. al., (2006). Imported OpenCyc terms and propositions comprise about 12% of the total Texai KB content of 12 million propositions. Content imported from WordNet version 2.1 constitutes about 20% of the total, and the remaining content was imported from the CMU Pronouncing Dictionary and from Wiktionary. Research by Reed and Lenat (2002) describes how the Cyc ontology accommodates the mapping of other ontologies, including WordNet, due to its wide coverage and expressiveness. In contrast to Cyc, the Texai KB does not have a sophisticated deductive inference engine, instead it uses only lookup, subsumption and context filtering. The Texai KB also lacks the majority of Cyc's natural language facilities.

Like Cyc, the Texai KB represents concepts as logical terms, which can be symbols, variables, numbers, character strings, named things, term formulas, propositions or rules. KB facts are logical propositions that each consist of a predicate and an argument list of related terms.

The following are the Texai KB objects used to represent knowledge:

Symbols are defined as uppercase character strings whose

first character is a colon. They do not represent a concept, but are rather used to designate formula placeholders. Examples: `:ARG1, :DEPTH`

Variables are defined as mixed-case character strings whose first character is a question mark. They appear only in rules as described below. Examples: `?SIT-TYPE, ?SOMEONE.`

Numbers are either 64-bit signed integers or 64-bit floating point. Examples: `-1, 0, 3.1416.`

Character strings are delimited by double quotes. For example: `"The creature of the protagonist in Mary Shelley's Frankenstein."`

Dates are strings in SQL date format. For example: `"2006-12-06".`

Time points are strings in SQL timestamp format. For example: `"2006-12-11 02:12:24".`

Named things are either atomic or non-atomic terms. Atomic terms are mixed-case symbols having the following naming convention: Types of things and named individuals begin with an upper case character. Predicate terms begin with a lower case character. Here are four example atomic terms:

```
Thing  Brazil TransportationDevice equals
```

Non-atomic terms consist of a functional relation with argument terms. They provide a way to compose new named things from existing concepts. The Texai KB renders formulas and propositions in the traditional mathematical syntax with the relation term preceding the argument list which is enclosed in parentheses and whose argument terms are separated by commas.

Here are two example non-atomic terms:

```
CollegeFootballTeamFn(BostonCollege)
OrganismPartTypeFn(Person, Liver)
```

The Texai KB operates under the unique names assumption in which named things are distinct. Accordingly, different terms by default do not represent the same concept unless explicitly related by an equality proposition. For example:

```
equals(MarkTwain, SamuelClemmens).
```

Note that RDF and OWL, in contrast, assume non-unique names.

Named things have the following three attributes, in addition to their term name and preferred lexical form:

- **Creator** is a term (usually atomic) representing the agent that created the term
- **Creation purpose** is a term (usually atomic) representing the process that created the term
- **Creation time point** is the term timestamp

Contexts are named things that partition the knowledge stored in the Texai KB. Each context term designates a region within one of many possible dimensions of context space. The foremost dimension of context space is the generality dimension that is arranged as a most-general to most-specific inheritance tree.

Term formulas consist of a property or function with an argument list. They are primarily used to represent the constituents of non-atomic terms and rules (described below).

Propositions are the most numerous Texai KB objects. They are also referred to as Ground Atomic Formulas (GAFs). Each one represents a ground (non-variable) fact in a specified context and is comprised of Creator, Creation purpose, and Creation time point (as defined above) and the following attributes:

- **Predicate** is the property term (usually atomic)
- **Argument list** is a list of non-variable terms, the first of which is usually an atomic term
- **Context** is the context term (usually atomic)
- **Strength** is a 64-bit floating-point number between -1.0 and 1.0 that ordinarily represents the degree to which the Texai KB believes the proposition to be true. A strength value of -1 indicates that the proposition is certainly false, a strength value of 0 indicates that the proposition's truth is unknown, and a strength value of 1 indicates that the proposition is certainly true.

An example proposition:

```
isa (TextualMaterial, StuffType)
```

Rules are quantified formulas having variables from which new facts may be deduced by yet-to-be-developed inference behavior. Each has the following attributes:

- **Logical formula** is a formula whose presentation format features infix logical operators, including **&&** (and), || (or) and **->** (implies), as well as prefix operators **!** (not), and the two quantifiers **forAll**, and **thereExists**.
- **Strength** as described above.
- **Creator**, **creation purpose**, and **creation time point** as described for named things.

An example rule:

```
(isa(?x, Integer) && greaterThan(?x, 0))
    -> isa(?x, PositiveInteger)
```

Java Source Code Annotation

Introduced as a part of Java version 1.5, the annotation facility offers a structured way to add type-checked attribute/value pairs to classes, class variables and methods. These annotations do not directly change the semantics of the Java source code, but external tools and the executing program itself, via reflection, can access the

annotations and perform appropriate operations. For example, the Java Persistence API defines the annotation **@Id** which is associated with a class variable to indicate that values of this variable should be mapped to the database column that uniquely identifies the persistent domain entity (i.e. its primary key). Application developers can extend the set of built-in annotations with their own annotation types.

Our research has led to a set of new annotations that declare KB class and property mappings and that consequently facilitate data exchange with the Semantic Web.

A Simple Example

The purpose of our semantic annotations is to declare, in the Java source code, the mappings between a domain entity (Java instance object) and a corresponding set of propositions (binary GAFs) in the Texai KB. Henceforth, we refer to propositions used that way as ***property assertions***. A domain entity contains fields and each field has one or more values. We persist each field value by using a property assertion whose predicate is specified by the field's semantic annotation, and whose first argument contains a reference to the domain entity, and whose second argument contains a field value:

```
predicate(domain-entity, field-value)
```

Note that this property assertion is equivalent to an RDF triple having this form:

```
domain-entity predicate field-value
```

A Java sample that illustrates semantic annotation is shown in figure 1.

```
...
@DomainContext(
  context="SampleDomainContext",
  subContextOf="BaseKB")
@DomainEntity
public final class Sample {
  @Id
  Long termId;
  @DomainProperty(identifier=true)
  final private String myString;
...
```

Figure 1: Semantic annotation of a sample Java class.

If `Sample001` is an instance of the `Sample` class, and that the field `myString` contains `"foo"`, then this domain entity field is persisted by the following property assertion in the Texai KB context `SampleDomainContext`:

```
myString(Sample001, "foo")
```

Domain entities are loaded from the Texai KB by first constructing an uninitialized domain entity, then querying its property assertions from the persistence context and using them to populate the domain entity's fields.

Annotations begin with the @ character and can optionally include a list of attribute/value pairs. Below are descriptions of the semantic annotations that are used in figure 1: @DomainContext, @DomainEntity, @Id and @DomainProperty:

@DomainContext
This annotation is applied to the domain package or class, and has the two following properties:

- ***context*** - This optional attribute defines the context from which the property assertions are loaded and to which they are persisted. Applied to a package-info.java file, the attribute designates the KB context for all the domain entities in that Java package. We recommend that each related group of domain entities have their own KB context.
- ***subContextOf*** - This optional attribute defines the super contexts of the given context. Each value must name an existing context in the KB, which is used when defining a new persistence context.

@DomainEntity
This annotation is applied to the domain entity class, and has the following properties:

- ***className*** - This optional attribute defines the name of this domain entity class and defaults to the qualified class name. It is used chiefly to map a domain entity to an existing Texai KB class that was imported from OpenCyc or somewhere else.
- ***typeOf*** - This optional attribute defines the types of this domain entity. Each value must name an existing class in the KB. Each value generates an "isa" assertion in the KB. This attribute should be specified for a new domain entity class.
- ***subClassOf*** - This optional attribute defines the super classes of this domain entity. Each value must name an existing class in the KB. This attribute should be specified for a new domain entity class.

@Id
This annotation is adapted from the Java Persistence API and is used to tag the Long-typed field that holds the domain entity identifier.

@DomainProperty
This annotation is applied to each of the domain entity fields and specifies the characteristics of the property assertion that persists the field's value(s). Domain property annotations neatly encapsulate information that otherwise would require a Relationship Object as described by Noble (1997). This annotation's nine attributes are described below:

- ***name*** - This optional attribute defines the name of the KB binary predicate that mapped to this association. The default value is the name of the annotated field.
- ***subPropertyOf*** - This optional attribute defines the more general property which has this property as a

specialization.

- ***domain*** - This optional attribute specifies the first argument type for this property assertion in the KB. The value must name an existing class in the KB. The default value of this attribute is the class of the domain entity that contains the field.
- ***range*** - This optional attribute specifies the second argument type for this property assertion in the KB. The value must name an existing class in the KB. The default value of this attribute is the class of the field's value.
- ***functional*** - This optional attribute defines whether the property is functional, having only one range value for a given domain value. The default value of this attribute is "false".
- ***inverse*** - This attribute defines whether the property is an inverse property with respect to the annotated field, in which case the field value is mapped to the domain of the property and the domain entity is mapped to the range of the property. The default value of this attribute is "false".
- ***identifier*** - This optional attribute defines whether the property is an identifier, having only one range value for a given domain value, and only one domain value for a given range value. When a domain property is an identifier, it is also functional. The default value of this attribute is "false".
- ***fetch*** - This optional attribute is adapted from the Java Persistence API and defines whether the value of the field or property should be lazily loaded or must be eagerly fetched. At present it is implemented only for collection valued fields. Allowed values are FetchType.LAZY, which is the default, and FetchType.EAGER. Because loading all the collection values of a persisted collection field can be time consuming, lazy loading is the more efficient alternative if there is low likelihood that the values will be accessed.
- ***strengthField*** - This optional attribute defines the associated double-typed field that contains the assertion strength, which ranges from [-1.0, ... +1.0]. If the domain property maps to a List-typed field then the assertion strength field may have the type List<Double>.

Domain Entity Operations

The framework class DomainEntityManager provides Create (persistence), Read, Update, and Delete (CRUD) operations.

Domain Entity Persistence Figure 2 is a Java code snippet that creates and persists the first domain entity (instance) of the Sample class.

```
final Sample sample =
  new Sample("Hello World");
domainEntityManager.persistDomainEntity(
  sample, creator, creationPurpose);
```

Figure 2: Code for domain entity persistence.

As a result, here are the property assertions created when the Sample instance is persisted. Note that the majority of these assertions define the class ontology. Subsequent persistence operations will only create assertions similar to numbers 6, 7 and 14.

1. **comment**(SampleDomainContext, "SampleDomainContext is a domain entity context.")
2. **isa**(SampleDomainContext, Microtheory)
3. **genlMt**(SampleDomainContext, BaseKB)
4. **isa**(org.texai.kb.persistence.sample.Sample, FirstOrderCollection)
5. **genls**(org.texai.kb.persistence.sample.Sample, Collection)
6. **isa**(org.texai.kb.persistence.sample.Sampl e_7274497, org.texai.kb.persistence.sample.Sample)
7. **domainEntityClassName**(org.texai.kb.persistence.sample.Sample_72 74497, "org.texai.kb.persistence.sample.Sample")
8. **comment**(myString, "myString(org.texai.kb.persistence.sample .Sample, CharacterString)")
9. **isa**(myString, FunctionalSlot)
10. **arity**(myString, 2)
11. **genlPreds**(myString, conceptuallyRelated)
12. **arg1Isa**(myString, org.texai.kb.persistence.sample.Sample)
13. **arg2Isa**(myString, CharacterString)
14. **myString**(org.texai.kb.persistence.sample.Sample_72 74497, "Hello World")

Domain Entity Load Domain entities may be loaded by an iterator. Figure 3 is a Java code snippet that loads a domain entity of the Sample class from an iterator over all the persisted instances in the Texai KB. This snippet omits the transaction handling statements.

```
final Iterator sampleIter =
  domainEntityManager.domainEntityIterator(
    Sample.class);
final Sample sample1 =
  (Sample) sampleIter.next();
```

Figure 3: Code for domain entity load via an iterator.

Domain entities may be loaded by their term id. Figure 4 is a Java code snippet that loads a Sample domain entity

given its term id. This snippet omits the transaction handling statements.

```
final Long termId = Long.valueOf(7274497L);
final Sample sample2 = (Sample)
  domainEntityManager.loadDomainEntity(
    termId);
```

Figure 4: Code for domain entity load via a term id.

Domain entities may be loaded by the atomic term that names an instance in the Texai KB. Figure 5 is a Java code snippet that loads a Sample domain entity given its representing atomic term. This snippet omits the transaction handling statements.

```
final AtomicTerm instanceTerm =
  domainEntityManager.findAtomicTermByTermNa
me(
    "org.texai.kb.persistence.sample.Sample_
7274497");
final Sample sample3 = (Sample)
  domainEntityManager.loadDomainEntity(
    instanceTerm);
```

Figure 5: Code for domain entity load via a given instance KB term.

Domain entities may be loaded by specifying a value object for an identifying property. Figure 6 is a Java code snippet that loads a Sample domain entity given the String value for the myString identifying property. An identifying property is a functional property that has only one domain entity associated with a unique value object. This snippet omits the transaction handling statements.

```
final AtomicTerm property =
domainEntityManager.
  findAtomicTermByTermName("myString");
final Object value = "Hello World";
final Sample sample4 = (Sample)
  domainEntityManager.
loadDomainEntityByIndentifyingPropertyValue(
    property, value, Sample.class);
```

Figure 6: Code for domain entity load via a given identifying property value.

Domain Entity Update Domain entities may be updated after loading them. Figure 7 is a Java code snippet that loads a domain entity of the Sample class from an iterator, and then persists the instance after modification.

```
final EntityTransaction entityTransaction =
  entityManager.getTransaction();
entityTransaction.begin();

// load via an iterator
final Iterator sampleIter =
  domainEntityManager.domainEntityIterator(
    Sample.class);
final Sample sample1 = (Sample)
   sampleIter.next();

// modify the domain entity
final String myString = "Hello World - " +
  UUID.randomUUID().toString();
sample1.setMyString(myString);

// persist the modified state to the KB
domainEntityManager.persistDomainEntity(
  sample1, creator, creationPurpose);
entityTransaction.commit();
```

Figure 7: Code for domain entity update.

Domain Entity Delete Domain entities may be deleted after loading them. Figure 8 is a Java code snippet that persists a domain entity of the Sample class and then deletes the instance after reloading it.

```
final EntityTransaction entityTransaction =
  entityManager.getTransaction();
entityTransaction.begin();

final String myString =
  UUID.randomUUID().toString();
final Sample sample1 = new Sample(myString);
domainEntityManager.persistDomainEntity(
  sample1, creator, creationPurpose);
entityTransaction.commit();

// reload via the identifying property value
entityTransaction.begin();
final AtomicTerm property =
domainEntityManager.findAtomicTermByTermName
("myString");
final Object value = myString;
final Sample sample2 = (Sample)
  domainEntityManager.
loadDomainEntityByIndentifyingPropertyValue(
    property, value, Sample.class);

// delete the persisted sample instance
domainEntityDeleter.deleteDomainEntity(
  sample2);
entityTransaction.commit();
```

Figure 8: Code for domain entity deletion.

Texai KB Implementation

The implementation of the Texai KB uses Enterprise Java Beans (EJB) version 3.0 which is the first EJB framework that supports Java annotation. There are several EJB vendors whose software is open source. This

implementation uses the freely available JBoss Application Server with EJB 3.0 extensions. The domain entity manager that loads, stores, creates and updates domain entities, is implemented as a set of EJB Session Beans. The J2EE container provides scalable and robust life-cycle support for the bean components, including dependency injection, which is defined as the automatic setting of references to system services and other application bean instances. As shown in figure 9, the system consists of modular layers which facilitate experimentation with implementation alternatives.

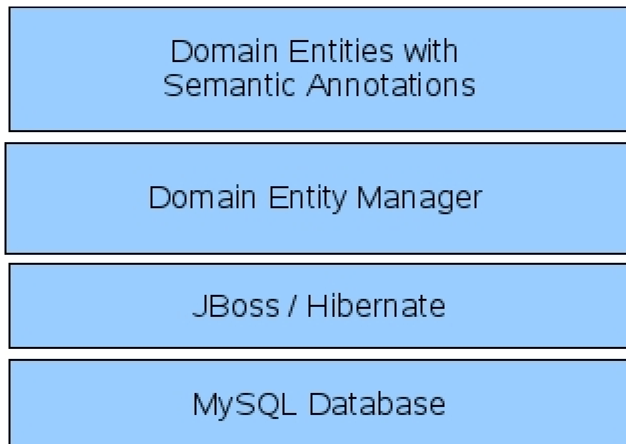

Figure 9: The Texai KB software stack.

The Texai KB application can be executed either within the container, using JBoss Application Server, or it can be executed as a simple application using a customized driver. The system is compatible with any database supported by JBoss/Hibernate. Currently, the Texai KB uses the open source MySQL database for the KB and persistent domain entities stored therein.

With regard to performance, domain entities can be persisted to the Texai KB at the rate of 200 propositions per second on an AMD 4800+ CPU using 64-bit Linux/Java/Hibernate/MySQL software. Each domain entity field requires at least one property assertion and consequently requires at least one database access per field for persistence. As a result, the performance with semantic annotations is slow in comparison with a non-semantic application that persists domain entities more efficiently to a relational database using one SQL statement for the whole object. However, to partially overcome this deficiency, the Texai KB uses Hibernate object caching to compensate when loading semantically annotated domain entities.

The Texai KB is open source software, licensed under the GPL, and all of its dependent libraries are also open source.

Related Research

RDFReactor (Völkel, M., 2006) is a Java code generator that takes as input an RDF or OWL schema and generates a set of domain classes. Each generated Java class models an RDF class and has field accessing methods for each property that is applicable to that class. In contrast to the Texai KB, which allows the semantic annotation of POJOs (Plain Old Java Objects), RDFReactor generates a built-in class inheritance hierarchy that follows the input schema.

ActiveRDF (Oren, E. et. al., 2007) is a lightweight Ruby application that can be easily adapted to a variety of RDF data sources. Its object manager provides an API to process RDF data as Ruby objects. Building Semantic Web applications is facilitated by using ActiveRDF with Ruby on Rails. Our approach is instead oriented towards the Java developer, and currently lacks a web-facing front end.

Future Work and Conclusion

The Texai KB and its existing set of semantic annotations have proved sufficient for an e-Science English dialog system under development. Future work will support persisting a greater variety of Java data types as applications require them. The current framework also eagerly retrieves all non-collection value objects when loading a persisted object. Hibernate, in contrast, uses Java byte code modification at runtime to lazily load even simple objects at their first access. The Texai KB persistence framework could be extended in the same fashion to more efficiently load persisted domain entities having large value object graphs.

The Texai KB is currently implemented with Hibernate and MySQL as the lowest architectural layer. It is possible to adapt other ORM and storage engines to host the KB content. Because current queries to find and load domain entities do not use the full power of SQL, it would be interesting to replace the Hibernate/MySQL layer with an alternative such as Oracle Berkeley DB Java Edition, which offers better performance in lieu of SQL compatibility.

In conclusion, this research has demonstrated that Java domain entities, semantically annotated during their initial development, can be persisted to a knowledge base which is implemented by a conventional database. The framework is only slightly more complex than ordinary EJB 3.0 object persistence but has the advantage of storing all data in Semantic Web-friendly format. Thus, the data may be easily accessed by other semantically based applications.

References

Bauer, C., King, G. Java Persistence With Hibernate, Manning Publications, 2007.

Frederickson, C.L. Object Mapping With Java Annotations. M.S. Thesis, Dept. of Computer Science, Montana State University, 2006.

Matuszek, C., Cabral, J., Witbrock, M., DeOliveira, J. An Introduction to the Syntax and Content of Cyc. In *Proceedings of the 2006 AAAI Spring Symposium on Formalizing and Compiling Background Knowledge and Its Applications to Knowledge Representation and Question Answering*, Stanford, CA, March 2006.

Noble, J. Basic Relationship Patterns. In *EuroPLOP Proceedings*, 1997.

Oren, E., Delbru, R., Gerke, S., Haller, A. and Decker, S. ActiveRDF: Object-Oriented Semantic Web Programming. In *WWW 2007* Banff, Alberta, Canada, May 2007.

Reed, S.L., and Lenat, D.B. Mapping Ontologies into Cyc. In *AAAI 2002 Conference Workshop on Ontologies For The Semantic Web,* Edmonton, Canada, July 2002.

Reed, S.L. Reference Manual for the Texai Knowledge Base v1.0, 2006, http://sf.net/projects/texai .

Sun Microsystems. The Java Persistence API - A Simpler Programming Model for Entity Persistence, 2006, http://java.sun.com/developer/technicalArticles/J2EE/jpa .

Völkel, M. RDFReactor – From Ontologies to Programmatic Data Access in *Jena Users Conference Proceedings, 2006.*

Yoder, J.W., Johnson, R.E., and Wilson, Q.D. Connecting Business Objects to Relational Databases. In *Proceedings of the 5th Conference on the Pattern Languages of Programs, Monticello-IL-EUA*, August 1998.