ActiveRDF: Object-Oriented Semantic Web Programming

Eyal Oren
eyal.oren@deri.org

Renaud Delbru
renaud.delbru@deri.org

Sebastian Gerke
sebastian.gerke@deri.org

Armin Haller
armin.haller@deri.org

Stefan Decker
stefan.decker@deri.org

Digital Enterprise Research Institute
National University of Ireland, Galway
Galway, Ireland

ABSTRACT

Object-oriented programming is the current mainstream programming paradigm but existing RDF APIs are mostly triple-oriented. Traditional techniques for bridging a similar gap between relational databases and object-oriented programs cannot be applied directly given the different nature of Semantic Web data, for example in the semantics of class membership, inheritance relations, and object conformance to schemas.

We present ActiveRDF, an object-oriented API for managing RDF data that offers full manipulation and querying of RDF data, does not rely on a schema and fully conforms to RDF(S) semantics. ActiveRDF can be used with different RDF data stores: adapters have been implemented to generic SPARQL endpoints, Sesame, Jena, Redland and YARS and new adapters can be added easily. In addition, integration with the popular Ruby on Rails framework enables rapid development of Semantic Web applications.

Categories and Subject Descriptors

D.2.12 [**Software**]: Software Engineering; Design Tools and Techniques[Object-oriented design methods]

General Terms

Languages, Design

Keywords

Semantic Web, RDF(S), object-oriented programming, scripting languages, Ruby on Rails

1. INTRODUCTION

The Semantic Web is a web of data that can be processed by machines, enabling them to interpret, combine and use Web data [4, p. 191]. RDF[1] is the foundational representation model of the Semantic Web. Statements in RDF are triples consisting of a subject, a predicate, and an object and assert that a subject has a property with some value.

Developing Semantic Web applications requires handling the RDF data model in a programming language. Although a majority of current software is developed in the object-oriented paradigm[2], programming in RDF is currently triple-based. For relational databases, object-oriented APIs have been long available: frameworks such as Hibernate [2] or ADO.Net provide an automatic mapping from relational databases to object-oriented programming.

Partially inspired by such object-relational mappings, the development of an object-oriented RDF API has been suggested several times [15, 16]. Such an API would map RDF Schema[3] (RDFS) classes to programming classes, RDF resources to programming objects and RDF predicates to methods on those objects, containing `Person.firstName` instead of `Resource.getProperty(http://xmlns.com/foaf/0.1/firstName)`.

In this paper we present an architecture and implementation of such an object-oriented RDF API. In section 2 we examine the differences between the object-oriented paradigm and the RDF model and explain why techniques used in object-relational mapping approaches are not sufficient. We present our solution architecture in section 3 and analyse the suitability of scripting languages for our mapping architecture. Section 4 introduces our implementation ActiveRDF, while section 5 illustrates the integration of ActiveRDF with the web application framework Ruby on Rails. We evaluate our work in section 6, discuss related approaches in section 7 and conclude in section 8.

2. PROBLEM STATEMENT

The conceptual model and semantics of RDF Schema differ substantially from the object-oriented paradigm, more so than the relational paradigm does. In this section we examine these differences and explain why existing mapping approaches do not suffice for Semantic Web data. Although the exact meaning of "object-oriented" varies [1], we will consider typical object-oriented features and focus mostly on Java.

2.1 Object–oriented versus RDF

The semantics of classes and instances in RDF Schema is open-world and description logics-based while object-oriented type systems are closed-world and constraint-based [10]. This

[1] `http://w3.org/RDF/`

[2] `http://www.tiobe.com/tpci.htm`

[3] `http://www.w3.org/TR/rdf-schema/`

WWW 2007, May 8–12, 2007, Banff, Alberta, Canada.
ACM 978-1-59593-654-7/07/0005.

fundamental semantic difference causes six mismatches:

1. *class membership:* in popular object-oriented languages such as Java or C# an object is member of exactly one class: its membership is fixed and is defined during the object instantiation. In RDF Schema, a resource can belong to multiple classes: its membership is not fixed but defined by its `rdf:type` and the properties that belong to the resource.

2. *class hierarchy:* in popular object-oriented type systems, such as in Java or C#, classes can inherit from at most one superclass, while in RDF Schema classes can inherit from multiple superclasses (including inheritance cycles).

3. *attribute vs. property:* in the object-oriented model, attributes are defined locally inside their class, can be used only by instances of that class, and generally have single-typed values. In contrast, RDF properties are stand-alone entities that can be used by any resource of any class and that can have values of different types.

4. *structural inheritance:* in object-oriented programming, objects inherit their attributes from their parent classes. In RDF Schema, since properties do not belong to a class, they are not inherited. Instead, property domains are propagated, but given their specific meaning indicating the class membership of resources using that property, domains propagate into the upwards direction of the class hierarchy.

5. *object conformance:* in most object-oriented languages, the structure of instances must exactly follow the definition of their classes, whereas in RDF Schema, a class definition is not exhaustive and does not constrain the structure of its instances: any RDF resource can use any property.

6. *flexibility:* object-oriented systems usually do not allow class definitions to evolve during runtime. In contrast, RDF is designed for integration of heterogeneous data with varying structure from varying sources, where both schema and data evolve during runtime.

2.2 Existing approaches

For relational databases several object–relational mappings exist, such as Java Data Objects and Hibernate for Java, ADO.Net for C# and ActiveRecord for Ruby. Most of these mappings follow the Active Domain Object or Active Record pattern [8, p. 160] which abstracts the database, simplifies data access and ensures data consistency.

Although the mapping frameworks differ in how they solve the impedance mismatch between the relational model, which is normalised for fast data retrieval, and the object-oriented model, which captures real-world objects as closely as possible, the general mapping is the same in all frameworks.

Tables are mapped to classes; table columns are mapped as attributes in the class, except for foreign keys which are mapped to object relationships; and every tuple in the relational model is mapped to an object. Intersection tables, which are introduced in the relational model to capture many-to-many relations, are mapped to object relationships (is-a, has-a relations).

To apply the general mapping methodology to RDF data, adjustments are required to address the six identified mismatches listed above. Existing approaches do not address these mismatches since they do not occur in relational data:

1. *class membership:* in the relational model, every tuple belongs to exactly one table, which maps without problem to the object-oriented requirement that every object must be member of exactly one class.

2. *class hierarchy:* a non-issue since no hierarchy of tables is allowed in the relational model.

3. *object attributes:* columns are the relational counterpart of object attributes, and map without problems to object attributes: columns are defined locally to a table, can be used only by tuples of that tables and have single-typed values.

4. *structural inheritance:* a non-issue since inheritance does not exist in the relational model.

5. *object conformance:* in the relational model each tuple must conform strictly to the table definition in the schema, which maps without problem to the object-oriented notion of object conformance. Although tables can have optional columns, a tuple cannot contain other columns than specified in the table definition.

6. *flexibility:* database systems are typically closed systems whose schema definitions do not change dynamically at runtime, rendering the level of flexibility needed for RDF data again a non-issue.

3. SOLUTION

The previous section discussed the mismatches between the object-oriented paradigm and RDF data and explained why existing object-relational mappings do not address these mismatches. To resolve these issues one can either impose restrictions on the usage of RDF or remove some restrictions in the object-oriented language.

We take the second approach: our solution is based on object-oriented scripting languages, where the mismatch between the object-oriented paradigm and RDF is smaller than with compiled object-oriented languages.

This section introduces scripting languages, explains their suitability and introduces our architecture for an object-oriented RDF API.

3.1 Suitability of scripting languages

Dynamic, general-purpose scripting languages such as Perl, Python, and Ruby are typically interpreted, use dynamic typing, have strong meta-programming capabilities (which enable the programmer to alter the semantics of the language) and allow runtime introspection [13]. Through dynamic typing and meta-programming, scripting languages enable us to implement a domain-specific language for RDF(S) data and alleviate the discussed mismatches as follows:

1. *class membership:* the dynamic typing of scripting languages does not require object types to be defined statically, these are rather determined at runtime by the capabilities of the object. Dynamic typing maps well to RDF(S) class membership which can also change

dynamically. Although objects in most scripting languages can have only one type at a time, we can override that behaviour using meta-programming.

2. *class hierarchy:* although most scripting languages do not support multiple inheritance, this behaviour can usually be changed through meta-programming.

3. *attribute vs. property:* the meta-programming facility of scripting languages enables the addition of attributes to objects dynamically whereas their dynamic typing enables the attributes to have values of multiple types.

4. *structural inheritance:* by itself, the lack of structural inheritance in RDF(S) does not form a problem for an object-oriented mapping. However, as we will describe in Sect. 4.1, the lack of structural inheritance limits the possibility of resolving ambiguous property shorthands, e.g. `Person.name`.

5. *object conformance:* scripting languages typically do not require objects to strictly conform to their class definitions but instead allow objects to deviate from their classes. For example, it is often possible to specify a different behaviour (method implementation) for several objects of the same class.

6. *flexibility:* since scripting languages are interpreted and do not rely on strict and prior-defined classes, they are well-suited for flexible environments in which both data and schema can evolve. The introspection allows programs to investigate the schemas and domain vocabulary that are available during program execution.

In summary, dynamic scripting languages offer the properties required for a virtual and flexible API for RDF(S) data. Our arguments apply equally well to any dynamic Turing-complete language with these capabilities.

3.2 Architecture

The general principle of our architecture is to represent RDF resources through transparent proxy objects. Each proxy object represents one RDF resource but does not contain any state. All methods (manipulations) on the proxy object are translated into (read or write) queries related to the proxy's RDF resource. Transparent proxy objects are simpler to implement than rich objects that copy the state and data of an RDF resource. Since rich objects often offer better performance, caching data in such rich objects can be implemented as an extension but requires a cache-management policy.

Our architecture consists of four layers, as shown in Fig. 1, that incrementally abstract RDF data into objects.

3.2.1 Object manager

The object manager is the library entry point and provides all the mapping functionality. It provides the domain model with all its manipulation and generic search methods. It is not a generated API (hence the name *virtual*), but uses meta-programming to catch unhandled method calls (such as `john.firstName`) and respond to them.

The object manager maps RDF data to objects and data manipulation to methods. For example, when the application calls a find method or when a new person is created,

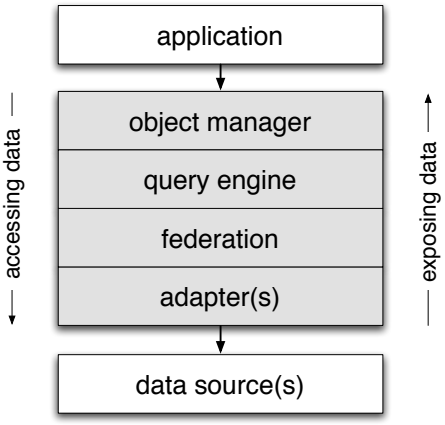

Figure 1: ActiveRDF architecture

the mapping layer translates this operation into a query on the data source. The object manager also creates object-oriented classes from RDF Schema classes if schema information is available.

Developers can augment the object-oriented classes with custom methods to provide additional behaviour. Such methods can be overridden in subclasses to define specific behaviour: for example, a `toString` method might return different results for different kinds of objects. In typical object-oriented systems, the definition in the most specific class is used when multiple method definitions are given.

However, given the multiple inheritance in RDF Schema and the possibility of multiple membership, an additional resolution strategy must be used for methods that are defined multiple times, in classes that have no inheritance relation to each other. Possible solutions are to execute the first-found method definition, to select the most applicable method through a more refined distance-measure, to let the developer explicitly indicate the definition to use, or to raise an error.

3.2.2 Query engine

The query engine provides an abstract query API that is independent of a specific data source and query language. It is used by the object manager to construct queries for each object manipulation.

3.2.3 Federation manager

The federation manager manages the collection of available data sources, distributes the queries over some or all of these sources and collects their results. The federation manager should, when querying multiple data sources, consolidate the results [7]: similar objects that are identified differently in the different data sources should be merged before the results are returned.

3.2.4 Adapters

Adapters provide access to a specific RDF data-store by translating generic RDF operations to a store-specific API. Such RDF data-store specific adapters are necessary, because of the absence of a general standardised query language which provides create, read, update, and delete access.

As such adapters are responsible for translating and exe-

cuting queries from the federation manager into a query language supported by their data source. Each adapter must implement a simple API, which allows new adapters to be added easily.

Adapters do not necessarily wrap RDF data sources, they could also wrap "legacy" sources such as desktop application data (as in the Aperture architecture [14]) or relational databases (as in the D2R system [5]), as long as they expose their query results as RDF.

4. IMPLEMENTATION

We have implemented the presented architecture in our Ruby library ActiveRDF, which provides a virtual API for managing RDF data in an object-oriented manner. We have reported on an initial implementation earlier [11]. Since then, ActiveRDF was completely re-implemented according to the architecture described above. ActiveRDF is currently implemented in around 600 lines of code; the adapters are written in on average 160 lines of code.

4.1 Object manager

The object manager offers a virtual API to manipulate RDF. This virtual API can be divided into three parts: mapping RDF(S) resources into objects, instance-level methods for manipulating these resources, and class-levels methods for searching resources.

In ActiveRDF every object can be member of many classes. Since Ruby does not allow such multiple membership, we override the built-in Ruby behaviour. All built-in methods that use the class of an object are overridden to rely on the `rdf:type`(s) in the data source.

Apart from the virtual API, developers can augment the domain model with custom methods. As discussed in section 3.2.1, a search strategy is needed to resolve multiple (clashing) method definitions in classes: as a pragmatic solution our current implementation uses the first-found definition.

4.1.1 Mapping resources

The object manager maps all RDF(S) classes to Ruby classes, all RDF resources to Ruby objects, and all RDF properties to attributes on the Ruby objects. All RDF resources are by default created as Ruby objects of class `RDFS::Resource`.

Domain-specific methods such as `john.age` or `john.name` are not generated but provided virtually: the object manager catches their invocation and translates the method call into a query. Without the object manager's interference, Ruby would throw a `MethodNotFound` error. Such metaprogramming caters for flexibility: as we do not generate the API but "pretend" it based on the data available at runtime, we do not need to recompile or regenerate the API when the data changes.

To prevent clashes between similarly-named classes in different libraries we map the RDF namespaces onto the namespace mechanism provided by Ruby. Listing 1 shows how to register a namespace abbreviation for the `FOAF` namespace and how to create an instance of `FOAF::Person`.

Listing 1: Mapping resources

```
Namespace.register(:foaf, 'http://xmlns.com/foaf/0.1')
ObjectManager.construct_classes
john = FOAF::Person.new('http://example.org/foaf.rdf#me')
```

4.1.2 Manipulating resources

Resources can be manipulated depending on the data access permissions and capabilities of the data source. Listing 2 shows how to use standard Ruby closure to traverse John's friends and print the name of each of them.

Listing 2: Traversing using Ruby closure

```
john.knows.each do |friend|
  puts friend.name
end
```

In this example, the object manager transparently catches the method calls `john.knows` and `friend.name` and translates each into a query. Part of this translation is determining the full URI of the predicate for "knows" and "name", which is straightforward with a unique local part, but ambiguous when different predicates have the same local parts. As discussed in Sect. 2, the schema definition cannot be used to determine which predicate might apply to a certain resource, since the schema does not constrain usage of predicates to classes. For example, every resource can use `foaf:name`, the resource then simply becomes of type `foaf:Person`. One might be tempted to use the schema definition and class hierarchy to limit this ambiguity and to find the most relevant property for a resource, but the RDF(S) notion of "domain" does not cater for this.

Developers can still use an ambiguous but convenient shorthand, as in Listing 3, but are not guaranteed the desired results since the first matching predicate will be used. Instead, they can explicitly specify the predicate through its namespace, as in Listing 4.

Listing 3: Ambiguous property accessors

```
john.knows.each do |friend|
  puts friend.name
end
```

Listing 4: Unambiguous property accessors

```
john.foaf::knows.each do |friend|
  puts friend.foaf::name
end
```

Each such resource manipulation is translated into a query. Invocations that change attribute values are handled similarly, but generate update queries instead of read queries.

4.1.3 Searching resources

If the URI of the resource is known to the application programmer, a proxy object is created as shown in the listing above. If the URI is not known, ActiveRDF offers two ways to search for it: with "dynamic finders" in the object manager, or through the Query API.

Listing 5 demonstrates the dynamic finders. The first shows a search returning all resources named "John", the second all thirty-year-olds named "John". These finders allow to locate a resource through one or more conjunctive clauses. If the developer requires more complicated queries the Query API can be used.

Listing 5: Dynamic finders

```
FOAF::Person.find_by_name('John')
FOAF::Person.find_by_name_and_age('John',30)
```

4.2 Query Engine

The current implementation of the query engine supports conjunctive datalog with select, distinct, and arbitrary where

clauses. Additionally it allows (for adapters that support this) counting query results, specifying the limit and offset of the query results and full-text keyword search. The query engine is used internally by the object manager for all manipulations. Further, it can be used by the application developer to execute complex queries on the data sources.

Listing 6 shows some typical queries. The first query counts the number of distinct predicates used in the dataset, the second one returns all distinct `foaf:name`s of the earlier defined John, and the third one finds all resources mentioning "apple".

Listing 6: Usage of Query API

```
Query.new.count.distinct(:p).where(:s, :p, :o)
Query.new.distinct(:o).where(john, FOAF::name, :o)
Query.new.distinct(:s).where(:s, :keyword, 'apple')
```

4.3 Federation Manager

The federation manager distributes queries amongst all registered data sources and aggregates their results. In the current implementation query distribution is achieved by simply querying all data sources sequentially; query result aggregation is achieved through a union of individual results (using either set union for distinct queries or bag union for non-distinct queries). We have not yet implemented a consolidation strategy in the federation manager, but we do offer an extension point for later addition of such functionality.

4.4 Adapters

Adapters wrap a data source into a standard interface, which includes methods such as `query, add` and `delete`. To translate the abstract query into the query language of the data source, adapters can either reuse existing translators or implement their own translation. To fully conform to RDF(S) semantics, ActiveRDF relies on the data sources to do so as well; data from sources without RDFS inferencing can still be used inside ActiveRDF, but the mapping will then not fully conform to RDF(S) semantics.

We have implemented adapters for generic SPARQL endpoints, to the RDF data stores Sesame [6], Jena [17], YARS [9], and Redland [3]. We have also implemented proof-of-concept adapters to desktop application data such as the Evolution email address book (exposed as FOAF data).

We have further developed rdflite, a simple and lightweight RDF store (and adapter) based on SQLite[4] with support for full-text search. We distribute rdflite as an adapter for ActiveRDF to enable simple prototyping without installing a fully-fledged RDF store.

5. SEMANTIC WEB ON RAILS

Ruby on Rails is a rapid application development framework for web applications, following the model–view–controller paradigm. Developers are presented with default models, views, and controllers and can adjust these to their domain. The model is usually provided by an existing database, the controller implements the control-flow in Ruby and the view is specified using HTML and embedded Ruby code.

Ruby on Rails has two main strengths: on the one hand it provides default application logic for the generic parts of web applications and several helper methods for data manipulation and JavaScript effects, alleviating developers from these tasks. On the other hand, since Ruby on Rails is targeted towards web applications that operate on relational databases, it integrates the business logic with the domain data using the ActiveRecord object-relational mapping: database tables serve as domain models and database tuples become Ruby instances.

We have designed ActiveRDF such that it can serve as a data layer in Ruby on Rails, replacing or augmenting the default ActiveRecord layer. As such, it provides a solution for rapid development of Semantic Web applications, leveraging the large and vibrant community of Ruby on Rails developers with their extensions and plug-ins. We have developed several web applications using ActiveRDF and Ruby on Rails; we will briefly describe two of them:

5.1 Semantic Conference Schedule

We have developed a simple conference schedule[5] purely built on Ruby on Rails and ActiveRDF. Originally developed for the International Semantic Web Conference 2006 this application operates on the conference metadata to show participants an overview of the conference schedule with details about each presentation and participant. The application operates strictly on RDF metadata using vocabularies such as ESWC, ISWC, SWRC, FOAF and ICAL.

Using ActiveRDF the integration of Rails with RDF data was straightforward and the development effort was minimal. Most development time was actually dedicated to support different browsers for the views. The models itself are automatically provided as virtual models, the controller (with all application logic) contains around 250 lines of code, and the views contain around 200 lines of HTML, Ruby and JavaScript code.

5.2 Faceted Metadata Browser

To allow navigation in arbitrary RDF datasets we have developed a faceted metadata browser[6]. Faceted browsing is a data exploration technique for large datasets. BrowseRDF extends this technique for complete graph-based data and adds algorithms to rank facets automatically based on facet entropy [12].

Again, using ActiveRDF the development effort was minimal once the formal model and the algorithms had been developed: the models are automatically provided, the controller contains around 300 lines of code, and the views contain around 250 lines of HTML, Ruby and JavaScript code.

BrowseRDF currently uses the rdflite data store, but the data source abstraction in ActiveRDF allows us to easily switch to a more scalable RDF store such as YARS or Sesame for larger datasets.

6. EVALUATION

We evaluate ActiveRDF in two ways: a quantitative evaluation to indicate possible performance overhead of our library and a qualitative evaluation to indicate the possible ease-of-use and increased productivity in software development. For practical reasons we have not measured productivity increase directly (as e.g. task completion speed of several similarly qualified programmers with and without ActiveRDF), instead an indication is given through the

[4] `http://www.sqlite.org`

[5] `http://schedule.semanticweb.org`

[6] `http://www.browserdf.org`

```
0  SELECT ?homepage WHERE { ?person <http://xmlns.com/foaf/0.1/name> Ashok Agrawala . ?person <http://owl.mindswap.org/2003/ont/owlweb.rdf
       #mindswapHomepage> ?homepage . }
1  SELECT ?p ?o WHERE { <http://activerdf.org/bnode#genid56> ?p ?o . }
2  SELECT ?p WHERE { <http://activerdf.org/bnode#genid56> ?p ?o . }
3  SELECT ?s ?p ?o WHERE { ?s ?p ?o . }
4  SELECT ?s ?p WHERE { ?s ?p ?o . }
5  SELECT ?s WHERE { ?s ?p ?o . }
6  SELECT DISTINCT ?p WHERE { <http://activerdf.org/bnode#genid56> ?p ?o . }
7  SELECT DISTINCT ?p WHERE { ?s ?p ?o . }
8  SELECT ?s WHERE { ?s <http://www.w3.org/1999/02/22−rdf−syntax−ns#type> <http://xmlns.com/foaf/0.1/Person> . }
```

Figure 2: Evaluation queries

relatively few lines of codes needed for the applications presented in section 5.

For quantitative evaluation, we compared query execution on Sesame (using various queries and various datasets) using the curl HTTP client (which shows the time needed by the data store for query answering), the Sesame Java API and ActiveRDF. We evaluated nine queries (ranging from selecting all triples to joins over two resources, see Fig. 2) using five different datasets (ranging from 2500–50.000 triples). Each test was first run to warm-boot the server and then repeated ten times. The tests were run on a server with two 1994MHz AMD Opteron 246 processors and 2Gb RAM.

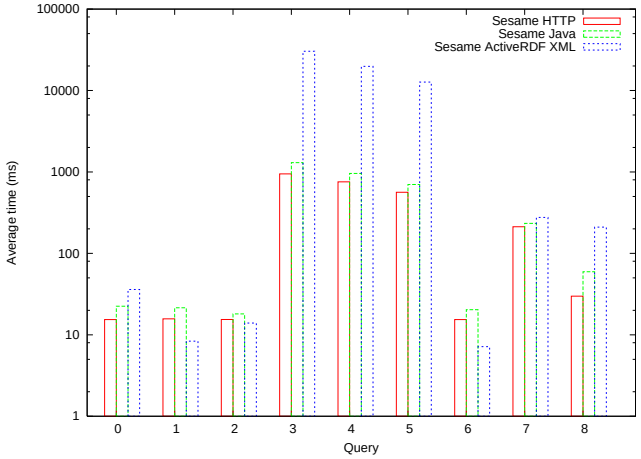

Figure 3: Querying Sesame in ActiveRDF

Fig. 3 shows the average response time (including result parsing in Java and ActiveRDF) of each query using curl, Java, and ActiveRDF in a logarithmic scale. It can be seen that for most queries ActiveRDF adds only little overhead. On some queries ActiveRDF seems to perform faster than using curl HTTP, which is probably due to random hardware variations and measurement difficulties in those small response time ranges.

For queries #3, #4 and #5 however the overhead of ActiveRDF is substantial. Because these queries return large amounts of XML results, we suspected the performance to be influenced by the Ruby XML parser. Fig. 4 therefore shows the average response time for same queries but with the JSON result format instead of XML: indeed the response time is on average halved for queries #3 (from $\pm 30s$ to $\pm 15s$), #4 (from $20s$ to $12s$) and #5 (from $13s$ to $7s$); note that the graphs are in logarithmic scale.

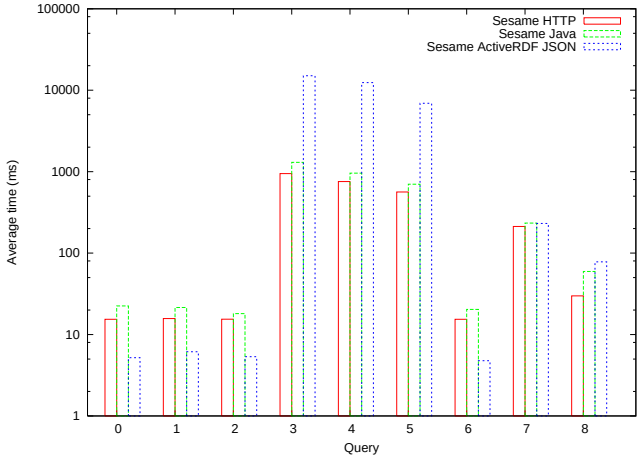

Figure 4: Querying Sesame ActiveRDF using JSON

7. RELATED WORK

Many RDF APIs exist currently[7] (in various programming languages). Some provide access to one specific RDF store, such as the Jena API [17] or the Sesame API [6], and some are agnostic to the underlying data store, such as RDF2Go[8]. Most of these APIs are generic and triple-based, offering methods such as getStatement and getObject. These are exactly the APIs that we want to abstract from.

The development of an object-oriented API has been attempted in Java in RdfReactor[9], Elmo[10] and Jastor[11]. These approaches ignore the flexible and semi-structured nature of RDF data and instead: (i) assume the existence of a schema, because they rely on the RDF Schema to generate corresponding classes, (ii) assume the stability of the schema, because they require manual regeneration and recompilation if the schema changes and (iii) assume the conformance of RDF data to such a schema, because they do not allow objects with different structure than their class definition.

8. CONCLUSION

We have presented ActiveRDF, an object-oriented library for RDF data written in Ruby. We have analysed why the techniques used in traditional object-relational mapping approaches are not sufficient for the Semantic Web and RDF in

[7] http://www.wiwiss.fu-berlin.de/suhl/bizer/toolkits
[8] http://rdf2go.ontoware.org
[9] http://rdfreactor.ontoware.org/
[10] http://www.openrdf.org/doc/elmo/users/index.html
[11] http://jastor.sourceforge.net/

particular. Based on a careful examination we have chosen to implement ActiveRDF in an object-oriented scripting languages. Among the advantages of these languages is the dynamic typing of objects, which maps well onto the RDF(S) class membership, meta-programming, which allows us to implement the multi-inheritance of RDF(S), and a relaxation of strict object conformance to class definitions.

ActiveRDF is light-weight and implemented in around 600 lines of code. It can be used with generic SPARQL endpoints, on popular RDF data stores, and with desktop application data. We have designed ActiveRDF such that it can serve as a data layer in Ruby on Rails, replacing or augmenting the default ActiveRecord layer, and providing a solution for rapid development of Semantic Web applications.

We have shown that ActiveRDF adds only little performance overhead, which can probably be decreased by carefully considering the parsing implementation. With its higher abstraction level and integration with Ruby on Rails, ActiveRDF allows the development of Semantic Web applications in relatively few lines of code.

Acknowledgements

The work presented in this paper was supported by the Science Foundation Ireland under Grants No. SFI/02/CE1/-I131 and SFI/04/BR/CS0694. We thank Benjamin Heitmann for his help in development and documentation, and the reviewers for their helpful feedback and suggestions.

9. REFERENCES

[1] D. J. Armstrong. The quarks of object-oriented development. *Communications of the ACM*, 49(2):123–128, 2006.

[2] C. Bauer and G. King. *Hibernate in Action*. Manning Publications, 2004.

[3] D. Beckett. The design and implementation of the Redland RDF application framework. *Computer Networks*, 39(5):577–588, 2002.

[4] T. Berners-Lee. *Weaving the Web – The Past, Present and Future of the World Wide Web by its Inventor*. Texere, 2000.

[5] C. Bizer and R. Cyganiak. D2R server – publishing relational databases on the Semantic Web (poster). In *Proceedings of the International Semantic Web Conference (ISWC)*. 2003.

[6] J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF Schema. In *Proceedings of the International Semantic Web Conference (ISWC)*, pp. 54–68. 2002.

[7] X. Dong, A. Halevy, and J. Madhavan. Reference reconciliation in complex information spaces. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 85–96. 2005.

[8] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.

[9] A. Harth and S. Decker. Optimized index structures for querying RDF from the web. In *Proceedings of the Latin-American Web Congress (LA-Web)*. 2005.

[10] A. Kalyanpur, D. Pastor, S. Battle, and J. Padget. Automatic mapping of OWL ontologies into Java. In *Proceedings of the International Conference on Software Engineering & Knowledge Engineering (SEKE)*. 2004.

[11] E. Oren and R. Delbru. ActiveRDF: Object-oriented RDF in Ruby. In *Proceedings of the ESWC Workshop on Scripting for the Semantic Web*. Jun. 2006.

[12] E. Oren, R. Delbru, and S. Decker. Extending faceted navigation for RDF data. In *Proceedings of the International Semantic Web Conference (ISWC)*. Nov. 2006.

[13] J. K. Ousterhout. Scripting: Higher-level programming for the 21st century. *IEEE Computer*, 31(3):23–30, 1998.

[14] L. Sauermann *et al.* Semantic desktop 2.0: The Gnowsis experience. In *Proceedings of the International Semantic Web Conference (ISWC)*, pp. 887–900. 2006.

[15] D. Schwabe, D. Brauner, D. A. Nunes, and G. Mamede. HyperSD: a semantic desktop as a semantic web application. In *Proceedings of the ISWC Workshop on the Semantic Desktop*. 2005.

[16] D. Vrandečić. Deep integration of scripting language and semantic web technologies. In *Proceedings of the ESWC Workshop on Scripting for the Semantic Web*. 2005.

[17] K. Wilkinson, C. Sayers, H. A. Kuno, and D. Reynolds. Efficient RDF storage and retrieval in Jena2. In *Proceedings of the International Workshop on Semantic Web and Databases (SWDB)*. 2003.