RDFReactor – From Ontologies to Programmatic Data Access

Max Völkel
FZI Forschungszentrum Informatik, Karlsruhe, Germany
voelkel@fzi.de

ABSTRACT

Developers used to object oriented programming have to make a paradigm shift in order to produce and manage Semantic Web data, e. g. RDF.

In this paper we describe the tool RDFReactor which transforms a given ontology in RDF Schema into a familiar, dynamic, object-oriented Java API – at the push of a button. Developers then are able to interact with java instances, thus allowing them to stay in their own world. The generated proxy objects contain no state and delegate all method calls to RDF model updates and queries.

RDFReactor potentially turns every Java developer into a Semantic Web application developer and enables them to use RDF correctly, efficiently and effectively without even knowing it. It is downloadable (GNU LGPL) at `http://rdfreactor.ontoware.org`.

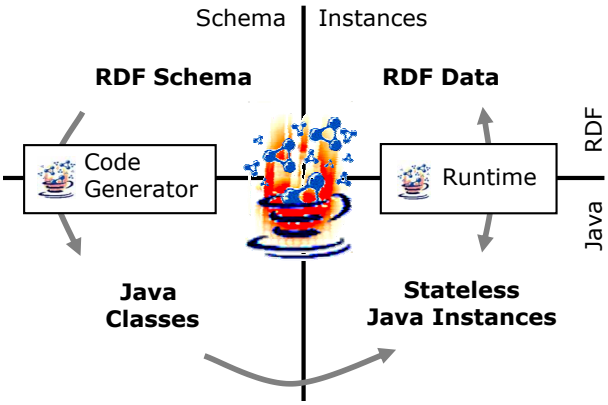

Figure 1: Mapping the two worlds

1. INTRODUCTION

A key promise of the Semantic Web is that of global interoperability, i. e. applications developed independent of each other will be able to read and use each others data. Ontologies are key enablers for the Semantic Web, they describe the semantics of data to enable ad-hoc interoperability. The Semantic Web is already rich in ontologies, but poor in applications that use semantic data. Why? Some evidence can be found by using Google queries which e. g. show millions of hits for "Java developer" and only hundred thousands of hits for queries like "ontology engineer". This might indicate a shortage of *ontology engineers*, who currently can also be seen as *developers* for ontology based applications.

Reuse of existing ontologies is crucial for efficiently and effectively reaching semantic interoperability on a global scale. Unfortunately, developers used to object oriented programming have to make a paradigm shift in order to produce and manage Semantic Web data, e. g. RDF [1].

All generated RDF data should be described by an ontology, in order to be usable. The task to make an existing Java application interoperable with the Semantic Web is a difficult task, as developers have to learn at the same time the RDF data model, RDF Schema syntax and semantics, and an API for model manipulation.

The main contribution of our work is to leverage the power and quantity of Java developers and Java tools for the Semantic Web by significantly reducing this burden. We introduce RDFReactor, a new open-source tool, which transforms a given RDF Schema ontology into an object-oriented Java API with domain-centric methods like `paper.setAuthor( Author a)` instead of `model.addTriple(...)`. This enables developers to interact with java proxy objects, thus allowing them to stay in their own world and at the same time to make use of the advantages RDF offers.

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

2. OVERVIEW

We distinguish two phases in application development: *API generation* followed by traditional *development*. An overview of the whole system can be found in Fig. 1: An RDF Schema (or an OWL ontology) is run once through the RDFReactor code generator which generates a number of type-safe, domain-specific Java classes. At runtime, instanced of these classes act as stateless proxies on the RDF model.

In the remainder of this Section we will first explain the features of the generated API and explain how these features were realised. Then, we explain features of the code generator and explain how those were realised.

2.1 Runtime Features

In this Section we explain the mapping from the object-oriented Java world to the triple-oriented RDF world. Basically, we map RDFS (or OWL) classes to Java classes and RDF properties to Java properties, accessed through `get()` and `set( value )` methods.

Additionally, every class generated by RDFReactor offers

the methods *Type*`[] getAll()`, `add(` *Type* `x)` and `remove(` *Type* `x)`, do deal with the multi-value nature of RDF properties. In fact, `get()` throws an exception if multiple values are present in the RDF model. The set-Method internally removes all values and then adds the given value.

Note that all API methods are type-safe. A developer using an RDFReactor generated API thus has full support from the Java compiler and it's IDE, e. g. auto-completion of method names.

At runtime, instances of the Java classes model resources in the RDF model. In the constructor, a reference to an RDF model and an identifier has to be supplied. We allow both for URIs or blank nodes to act as identifiers. There are no limitations with respect to blank node handling. For the RDF model manipulation, we use an abstraction layer, RDF_2Go[2], which has adaptors to Jena[1], Sesame, YARS and others.

One of the key advantages of using an RDFReactor generated API is synchronicity: Each method set or add requests is translated into an add statement operation, each get or getAll method call is executed as a query to the RDF model. All java instances are completely stateless, they act like proxies for RDF resources. Additionally, the API offers the following set of operations:

SPARQL: The `Bridge` class supplies a method to perform arbitrary SPARQL SELECT queries and return the result as type-safe Java domain classes.

List support: We offer methods to transform rdf:List constructs to Java lists and vice vers.

Map support: Each class generated by RDFReactor additionally implements the Java Collections Map interface, mapping properties to keys and the objects of RDF statements to Map entry values. Again, all calls are transparently translated into triple update and query operations, no state is duplicated at any time.

Inverse properties: As properties in RDF often hold references to other objects it makes a lot of sense to query also in the other direction. Instead of asking "whom do I know?" it is also intersting to find out "who knows me?" on the level of RDF resources. Thus the generated classes offer methods to get a list of backwards related entities. For each property p relating instances of class A with class B, the class A has a method to get, set, add and remove values as explained above. The class B has a method *Type*`[] get`p`_Inverse()` to get all instances of class A related to B via p.

Typing: The Java keyword `instanceof` cannot reflect schema or type changes due to the static nature of the Java language. As it is a keyword it can not even be overwritten with advanced method such as dynamic proxies. Instead, the method `Class[] getAllType()` returns a list of all RDF or OWL classes this instance of *currently* type of.

Cardinality constrains: OWL cardinality constraints are reflected in the API: `add` raises a `CardinalityException` if more elements than allowed are added. Similarly, `remove` raises an exception if a min cardinality constrains would be violated. The cardinality constraints are modelled statically in the generate code and do not reflect schema changes. This allows to generate methods that raise no exceptions for properties without constraints. Such methods are easier to use for developers.

Triple Store Independent. All generated APIs access RDF models only through RDF_2Go, thus the generated API can be ported without changes to another triple store.

[2] `http://rdf2go.ontoware.org`

2.2 Code Generator Features

RDFReactor translates an ontology into a number of domain-specific classes, which are explained in the previous Sections. In this Section, we explain features of the code generator.

The developer creates or, preferably, reuses an RDF Schema or OWL ontology as the backbone of the application. We map ontology classes to domain-specific Java classes and relations to properties, as explained in Sec. 2.1 and Sec. 3. The code generator employs a number of features to help an ordinary Java developer to become a semantic web developer (without actually having her to explain anything beyond URIs in most cases):

- The Java classes and methods are in most cases named according to usual Java coding conventions, even if no labels are present in the ontology.
- RDFReactor works both with RDF Schemas and OWL ontologies, respecting the ontology semantics (i. e. reasoning is used).
- After code generation, the API is ready to use. It's fully implemented and documented (JavaDoc).
- Multiple inheritance is resolved, a subclass tree is extracted to produce more readable code.
- Multiple domains and ranges are handled in an appropriate, type-safe manner.
- Changes to the generated source code are easy: Most methods are implemented with a single line of code. If frequent schema changes are expected, customized subclasses should be used, so that changes don't get lost when the code generation is triggered again.
- Cardinality constrains are reflected in the API and checked at runtime.
- The generated source code style is easy to change, as it is generated with a template language.

In the next Section we explain *how* these features were designed.

3. RUNTIME DESIGN

A key of the runtime design is a subclass hierarchy modelling ontological layering in Java. The lowest layer mediates between RDF's graph model and object-oriented concepts. We now explain the layering in detail:

3.1 OO-Graph mapping layer

This layer contains hand-written methods to translate from object-oriented method calls into triple-centric methods.

ReactorBase is the key to understand RDFReactor concepts. It acts as a generic RDF resource, without any RDF or RDFS semantics beyond the triple model. Each generated Java class directly or indirectly inherits from ReactorBaseImpl. Each ReactorBaseImpl instance knows it's URI (or blank node) and the RDFS Class it represents in the API. Java objects are converted to the correct RDF types (e. g. URI, xsd:int, ...). Multiple ReactorBase instances looking at the same resource can have different RDFS Classes, of course, allowing for different perspectives on the same resource.

ReactorBase also implements a `Map<URI, Object>` interface, allowing *all* RDFReactor instances to be used additionally in this generic fashion.

3.2 Ontology language layer

On this and lower layers, URIs have semantics. In our case, RDF, RDFS, OWL and XSD are considered. Classes like *Resource* and *Class* offer methods to access typical RDF(S) properties such as labels (`rdfs:label`), comments (`rdfs:comment`), types (`rdf:type`) and subclasses (`rdfs:subClassOf`). These classes have been generated by applying the Code Generator to the RDF Schema definitions of RDFS and OWL. In fact, the auto-generated methods for the property *type* in the class *Resource* are used as a substitute to Java's `instanceof`, as explained above.

Note how RDFReactor can operate on RDFS's class and property structures without running into any meta-modelling issues. This simply due to the fact that all RDFS and OWL semantics are handled in the underlying store.

3.3 Domain layer

This layer contains the code generated from a particular RDFS or OWL ontology, e. g. *Person* or *Image*. They inherit from the class *Class* of the layer above.

In order to illustrate the layered design, we give now some examples. We could use the ReactorBase implementation directly and invoke it as follows:

```
1: URI FoafPhone = new URI(
     "http://xmlns.com/foaf/0.1/phone");
2: URI FoafKnows = new URI(
     "http://xmlns.com/foaf/0.1/knows");
3: ReactorBase heiko =
     new ReactorBaseImpl("urn://heiko");
4: ReactorBase max   =
     new ReactorBaseImpl("urn://max");

5: heiko.add( FoafPhone, 123 );
6: heiko.add( FoafKnows, max );

7: int i = (int) heiko.get(
     FoafPhone, Integer.class );
8: Person[] heikosFriends =
     (Person[]) heiko.getAll(
     FoafKnows, Person.class );
```

The lines 5 and 6 immediately results in the triples `{ <urn://heiko> foaf:phone "123"(xsd:int)}` and `{ <urn://heiko> foaf:knows <urn://max>}` to be added to the model. This is nice, but RDFReactor can do better by generating type-safe domain-specific classes (which is assumed in line 8). Let's assume we generated a class *Person* from a FOAF-like ontology [2]. It will look as follows:

```
class Person extends ReactorBaseImpl
```

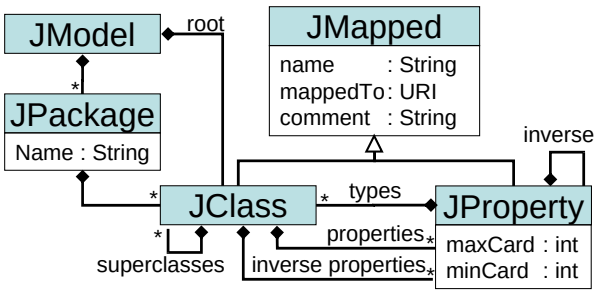

Figure 2: JModel - the internal ontology model

```
public void addPhone( int value ) {
  add( FoafPhone, value );
}

/**
 * @throws RDFDataException
 * if multiple values found
 */
public int getPhone()
throws RDFDataException
  return (int) get( FoafPhone );
}

public void addKnows( Person p ) {
  add( FoafKnows, p);
}

public Person[] getAllKnows() {
  return (Person[]) getAll(
    FoafKnows, Person.class );
}
```

As you can see, the domain specific classes are merely a customising ReactorBaseImpl by hiding the type conversions. This reflects well the strategy employed in the design of RDFReactor: We separate the concern of graph-OO-mapping (ReactorBase, Bridge) from the concern of mapping loosely-typed-RDFS to strictly-typed Java (generated code). The domain-specific class can be used by ordinary Java developers without any knowledge about RDF as follows:

```
Person p = new Person("urn://heiko");
p.setAge(29);
int i = p.getAge();
Person[] heikosFriends =
  heiko.getAllKnows();
```

We currently have support for the following datatypes as values in domain specific methods (get, set, ...): all ReactorBase instances, URI, URL, String, DatatypeLiteral, LanguageTagLiteral. Additionally with correct XSD mapping: Integer, Long.

4. CODE GENERATOR DESIGN

First, we translate the ontology into an internal model (`JModel`). The internal model (c. f. Fig. 2) looks as follows:

JModel models a set of *JPackages*. It is mapped to an RDF model.

JPackage models a package in Java, e. g. ”org.ontoware.rdfreactor.foaf´´. Modelling this explicitly allows to map different RDF namespaces to different Java packages. A *JPackage* has a name and a list of classes.

JClass has a name (e. g. "Person´´), a URI it is mapped to, a comment, a list of properties, a list of inverse properties and a list of superclasses.

JProperty has a name (e. g. "phone´´), a URI it is mapped to, a comment, a list of types (*JClasses*), a minimum and maximum cardinality and an inverse *JProperty*.

The JModel is very expressive and allows many things that are illegal in Java, e. g. class cycles, multiple inheritance and properties with no type. Algorithms to make the JModel stricter are successively applied, allowing for reuse regardless of source ontology language. Finally, the JModel is strict enough to generate source code with the help of a template processor. In the remainder of this Section we explain the code generation process in detail. In the future, we consider replacing the JModel with the Ecore framework from Eclipse or other works based on the *Ontology Definition Metamodel* [3].

Load built-ins. We initialise a JModel with the built-in mappings of URIs to the Java classes of the schema layer (c. f. Sec. 3). Of course, the default model is different for RDFS and OWL. A new JPackage for the API to be generated is created and added to the JModel.

Inferencing. RDFS and OWL inferencing is applied to the source ontology. All handling of ontology language semantics is thus delegated to existing implementations.

Inspect Classes. For each ontology class *oc*, we distinguish three cases. First, if it is an OWL restriction and thus not a *real* class: ignore it. Second, if the JModel already contains a JClass mapped to the URI of *oc*: ignore it as well. Otherwise we add a new JClass to the JPackage. In order to find a legal Java class name, a number of little things has to be done, see Sec. 4.1.

If the ontology class has an rdfs:comment it is copied over in the JModel. Later is is integrated with the auto-generated JavaDoc of the class.

Multiple Inheritance. Inheritance is tackled in two steps: First, each superclass relation in the ontology is mapped to a superclass relation in the JModel. Second, the inheritance hierarchy is flattened, to remove multiple inheritance, as described in Sec. 4.2.

Inspect Properties. For each property *op* in the ontology: In the JModel, we look up the JClasses mapped to the IDs (URI or blank node) of *op*'s domains. For each domain d we generate a JProperty jp and attach it to the respective JClass. The JProperty is named and commented with the same strategy as classes (c. f. Sec. 4.1).

Then, for each range relation of the property *op*, we add the corresponding JClass as a type of the property jp. Multiple types for a property are handled in later steps of the code generation.

Inverse Properties. For each JProperty jp with domain d in the JModel, we add inverse properties: For each JClass r in the range of jp we add an inverse property 'jp_Inverse' to r with the range d.

Source Code Generation. Finally the JModel is suitable for a template-based code generation. Emitting source code with a template language allows better control over the generated syntax, especially indentation. We initialise a Velocity[3] template and use it to generate the source code of a Java class for each JClass in the JModel. The template has about 300 lines of codes and is described in detail in Sec. 4.3.

The overall process of code generation can be summarised as transformation from source ontology language into an internal representation (JModel), stepwise simplification of the internal model until a template-based code generation approach can emit the final source code. In the remainder of this Section we explain in detail issues regarding naming, multiple inheritance and the template.

4.1 Naming

In order to name a class or a property adequately, a number of strategies are used:

First, a raw name is created by looking at the rdfs:label in the schema (e. g. 'IM-account'). If no label was found, the part of the URI after the fragment identifier ('#') is used (e. g. 'imAccount'). If no such delimiter is present in the URI, the part after the last slash ('/') is used. This name is then transformed into a legal Java identifier as follows: Spaces and underscore are removed. Then all non-alphanumeric letters are converted into an underscore (e. g. 'IM_account'). If the resulting name *starts* with a number, the name is prefixed with 'a_'. Finally, the first letter is capitalised, to reflect common Java naming conventions (e. g. 'ImAccount').

The resulting name is compared with a list of taken names, to avoid name clashes. If such a conflict occurs, a longer name is created from the URI. If the part before the last number sign or slash is known, the correct namespace prefix is used (e. g. 'foaf', resulting in 'FoafImAccount'). Otherwise the last part of the URI consisting only of letters is used (e. g. from the URI 'http://xmlns.com/wordnet/1.6/' the prefix 'wordnet' would be extracted, as '1.6' is not consisting only of letters, resulting in 'WordnetImAccount'). If that name is *also* taken already in the schema at hand, a fallback to transform the full URI into a legal Java identifier is used. As it turns out, naming is less simple then one might think. For properties, removing leading parts such as 'is ', 'has ', 'is_', 'has_' leads to methods names better reflecting common Java API design.

4.2 Inheritance Flattening

For each class in the JModel the following cases have to be considered:

- If the class has no superclass, the root class of the JModel (either rdfs:Class or owl:Class) is set.
- Classes with exactly one superclass are left unchanged.
- If multiple superclasses are found, the superclass that has itself most superclasses is used. Note that setting the JModel root class for all classes as the superclass would result in no loss of functionality in the generated API. The whole point of modelling ontology subclass

[3]Jakarta Velocity, a template engine. See `http://jakarta.apache.org/velocity/`

relations at least partially in Java is the desire to generate code that is simpler to change manually later on. There are other options to extract a strict tree from the inheritance graph in the ontology, though.

4.3 Template

The template has to take care of a number of things in order to produce a good source code. In particular the template generates the following:

Minimal Header. A package declaration, import statements for basic RDF classes and conditional imports, e. g. for exceptions. Only classes that are used are imported, to avoid having IDEs such as Eclipse to bug the user with warnings.

Binding. Each class is bound to an specific ontology class with a URI constant.

Documentation. An overview comment that explains which properties are handled in this class is also added to the JavaDoc. Each class and method are also documented.

Constants. All URIs that are used in the body of the class are defined as constants.

Constructors. A number of constructors allowing for instances to be created from a URI, URL, blank node or no identifier (a random URI will be used). Some protected constructors are also generated, to allow constructor delegation up to the top (ReactorBaseImpl).

Properties. For each property, methods are generated *according to cardinality constraints and number of given ranges.* If no cardinality constraints have been state *or* the maximum cardinality is exactly 1, a get and set methods are created. A remove method, encoding cardinality constraints in the implementation as needed (e. g. 'remove(value, 3)', it a minimum of three elements must be guarded), is generated. If the maximum cardinality is greater than 1 or not set, add and getAll methods are generated.

Multiple ranges result in multiple set or add methods with the same name but different types, exploiting method overloading. For get or getAll methods, it is not possible to have the same method with different return types. Thus we disambiguate method names with a postfix of '_asTYPE' to indicate the desired return type.

Lean code. All generated methods are implemented (typically in single line of code, thanks to the inheritance), documented (in JavaDoc) and throw appropriate exceptions (as documented in the JavaDoc) if things go wrong.

Queries. A method to return all instances of the generated class for a given model.

Inverse Properties. For each inverse property of the class, get and add methods.

4.4 Implementation Details

First we wrote the ReactorBaseImpl class and hand-coded essential parts of the schema layer (c. f. Sec. 3). Then we wrote the code generator, making use of the schema layer classes (e. g. `getLabel()`). Then we ran the preliminary code generator on the RDF Schema of RDF Schema to get a better (more features, better documented) schema layer. The new schema layer classes were then used to enhance the code generator. This way of bootstrapping seems well suited for other programming languages as well.

As a second remark, the current handling of primitive data types works only thanks to the new feature autoboxing and -unboxing introduced in Java 1.5. Autoboxing converts seamlessly from primitive types and their corresponding Classes, e. g. `int` and `Integer.class`.

5. MODEL MANIPULATION DESIGN

In order to stay independent of a specific triple store and be able to offer RDFReactor APIs on top of a number of stores, we developed the abstraction layer RDF$_2$Go. RDF$_2$Go is a simple wrapper over Java triple (and quad[4]) stores, similar in spirit of Jakarta Commons Logging. It allows developers to un-tangle their semantic web applications from a specific store and thus profit from using always the best store (performance, features). Writing new adaptors is easy: only few methods have to be implemented.

RDF$_2$Go currently[5] comes with adaptors for Jena 2.2, Jena 2.3, Sesame 1, Sesame 2, YARS, NG4J and even an experimental adaptor for the *Java Content Repository* API (JSR170). The core features of RDF$_2$Go are:

- add and remove triples with one-method-call, no costly object creation involved
- query for triple patterns
- SPARQL (although not all stores support it)
- full support for quad stores
- built-in URIs for RDF, RDFS, XSD
- No support for parsing. Users should use the supplied parser from the triple stores instead. We don't re-implement everything.

RDF$_2$Go maps URIs to `java.net.URI`, blank nodes to `org.ontoware.rdf2go.BlankNode`, plain literals simply to `java.lang.String`, language tagged literals to `org.ontoware.rdf2go.LanguageTagLiteral`, and data-typed literals to `org.ontoware.rdf2go.DatatypeLiteral`. This allows to add statements as simple as

```
// assume some URIs have been created
URI mike = new URI("urn://mike"); URI foafName = new
URI("http://foaf.com#name");

// add triple
model.addStatement( mike, foafName, "Mike Müller" );
```

Internally, RDFReactor relies on Jena 2.3 (accessed through RDF$_2$Go) in the code generator (c.f. Sec. 4.

[4]Quad stores store quads, not only triples. This is also known as *Named Graphs*

[5]release downloads, including all libraries: `http://ontoware.org/frs/?group_id=37&release_id=173`

There are other RDF triple abstraction layer approaches, but to our knowledge none is as simplistic and comes with such a large number of adapters. Related approaches to RDF_2Go are KPOntology [6], Trippi [7] and of course the Melnik API [8].

6. RELATED WORK

There have been several projects like RDFReactor, that attempted to map RDFS to Java. The predecessor of RDFReactor was *OntoJava* [4] from AIFB. OntoJava was an early approach and lacks basic features such as multiple inheritance, RDF resources with more than one RDF literal related to it or RDF properties with multiple domains.

Rdf2Java [9] overcomes the inability of RDF Schema to express *restrictions* by using the *Protégé* [6] annotations. RDF Schemas created with *Protégé* can therefore be used to generated constrained Java classes. Rdf2Java generates Java classes that act as a facade to a Jena model.

An approach to map OWL full to Java is described in [5]. Here the multiple inheritance problem, which applies to RDF Schema as well, is solved by using Java interfaces. In Java, a class can have only one superclass, but may implement many interfaces. Unfortunately, the OWL type system is only present in the form of raised exceptions, as all properties are mapped to methods that get and set generic non-type-safe Java.util.List methods.

Summary: The current mapping approaches create custom Java classes in source code. RDFReactor is not tied to a particular triple store, allows for easy changes in the source code via Velocity templates and handles inverse properties in a unique way.

7. USAGE EXAMPLE

Then intended way of using RDFReactor involves two parts: API building and API usage.

API building involves some choices by an RDF-aware developer. First, an existing ontology is re-used and possibly adapted, or a new ontology is designed. Second, the API is generated by running RDFReactor on the ontology. Finally the generated classes are bundled together with the RDFReactor runtime, a triple store, and an RDF_2Go adapter for this store. The resulting package can be distributed.

API usage can be done by any Java programmer, with little or no knowledge about RDF, ontologies or the Semantic Web. The developer simply instantiates the generated classes and uses them, e. g. for developing a graphical user interface for a specific domain. All user interaction with the final UI will result in correct RDF, without any extra effort on the developer side. And this RDF is described by the ontology, which has been used to generate the the API.

RDFReactor has been used in the open-source RDF-versioning system SemVersion[10]. SemVersion in turn is used in the MarcOnt project[11]. Most of the internal API has been generated from a custom RDF Schema. Some parts of RDFReactor have been generated by generating APIs for RDF Schema and OWL. These APIs are now used by the Code Generator.

In an ideal world, tools similar to RDFReactor, such as ActiveRDF[12] for Python, would exist for all major programming languages and allow other developers to use the RDF data in an effortless way.

8. CONCLUSION AND OUTLOOK

In this paper we have shown how a domain-centric, usable Java API can be generated from an arbitrary RDF Schema. Our implementation, RDFReactor, is due to it's dynamic nature always in-sync with the RDF data model. Each class inherits from `ReactorBase`, which allows the developer to manipulate arbitrary RDF properties directly (`set(URI prop, Object o)`. Thus we do not restrict the expressivity of RDF in any way. Additionally, code generated by RDFReactor is fully customisable, thus method names and URIs can be changed easily.

In the future, we plan to add transaction support to RDF_2Go and RDFReactor. The handling of OWL will be enhanced.

We help to make the ontology reuse promise a reality by enabling the average Java developer to consume and produce data conforming to existing ontologies through domain-specific Java APIs. The main advantage of our approach is that developers who use the generated API don't have to know RDF at all, but can make full advantage of its' capabilities. RDFReactor is downloadable (Open Source GNU LGPL) at `http://rdfreactor.ontoware.org`.

Acknowledgements. This research was partially supported by the European Commission under contract FP6-507482 (Knowledge Web)and FP6-027705 (Nepomuk). The expressed content is the view of the authors but not necessarily the view of the Knowledge Web Network of Excellence as a whole. Many thanks to Andreas Eberhart, Daniel Oberle, Sudhir Agarwal, Peter Haase, Heiko Haller and Reviewer-17.

9. REFERENCES

[1] J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson. Jena: implementing the semantic web recommendations. In S. I. Feldman, M. Uretsky, M. Najork, and C. E. Wills, editors, *WWW (Alternate Track Papers & Posters)*, pages 74–83. ACM, 2004.

[2] L. M. Dan Brickley. Foaf vocabulary specification, 04 2004.

[3] I. S. S. DSTC, Gentleware. Ontology definition metamodel.

[4] A. Eberhart. Automatic generation of java/sql based inference engines from rdf schema and ruleml. In *Lecture Notes in Computer Science*, volume 2342, 01 2002.

[5] A. Kalyanpur, D. Pastor, S. Battle, and J. Padget. Automatic mapping of owl ontologies into java. In *Proceedings of SEKE 2004*, Banff, Canada, June 2004.

[6] N. F. Noy, M. Sintek, S. Decker, M. Crubézy, R. W. Fergerson, and M. A. Musen. Creating semantic web contents with protégé-2000. *IEEE Intelligent Systems*, 16(2):60–71, 2001.

[6] `http://kpontology.isoco.com/download.html`
[7] `http://trippi.sourceforge.net/`
[8] `http://www-db.stanford.edu/~melnik/rdf/api.html`
[9] http://rdf2Java.opendfki.de
[10] `http://semversion.ontoware.org`
[11] `http://www.marcont.org`
[12] `http://activerdf.org/`