Jena: Implementing the Semantic Web Recommendations

Jeremy J. Carroll*
Dave Reynolds*
*HP Labs, Bristol
UK

Ian Dickinson*
Andy Seaborne*
firstname.lastname@hp.com

Chris Dollin*
Kevin Wilkinson†
†HP Labs, Palo Alto
CA. USA

ABSTRACT

The new Semantic Web recommendations for RDF, RDFS and OWL have, at their heart, the RDF graph. Jena2, a second-generation RDF toolkit, is similarly centered on the RDF graph. RDFS and OWL reasoning are seen as graph-to-graph transforms, producing graphs of virtual triples. Rich APIs are provided. The Model API includes support for other aspects of the RDF recommendations, such as containers and reification. The Ontology API includes support for RDFS and OWL, including advanced OWL Full support. Jena includes the *de facto* reference RDF/XML parser, and provides RDF/XML output using the full range of the rich RDF/XML grammar. N3 I/O is supported. RDF graphs can be stored in-memory or in databases. Jena's query language, RDQL, and the Web API are both offered for the next round of standardization.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures, D.3.2 [**Programming Languages**]: Language Classifications – Java, I.2.4 [**Artificial Intelligence**]: Knowledge Representation Formalisms and Methods – *Representation languages.*

General Terms

Standardization, Languages.

Keywords

RDF, OWL, Jena, RDQL, Semantic Web

1. INTRODUCTION

The new recommendations for the Resource Description Framework (RDF) and the Web Ontology Language (OWL) have just been published. They provide a simple triple-based representation of knowledge, with formal semantics allowing for automated inference. RDFS and OWL also provide some useful vocabulary, particularly for building schema and ontologies.

1.1 What is Jena?

Jena is a leading Semantic Web toolkit [22] for Java programmers. Jena1 was first released in 2000 and has had over 10,000 downloads. Jena2, with a revised internal architecture and many new features, was released in August 2003, and has had over 7,000 downloads. This paper presents Jena2, concentrating on the key architectural and design principles.

The heart of the Semantic Web recommendations is the RDF Graph [20] as a universal data structure. An RDF graph is simply a set of triples (S, P, O), where P names a binary predicate over (S, O). Jena2 similarly has the Graph as its core interface around which the other components are built.

WWW 2004, May 17–22, 2004, New York, New York, USA.
ACM 1-58113-912-8/04/0005.

The main contribution of Jena1 [22] was the rich *Model* API for manipulating RDF graphs. Around this API, Jena1 provided various tools, including I/O modules for: RDF/XML [9], [10], N3 [7], and N-triple [13]; and the query language RDQL [25]. Using the API the user can choose to store RDF graphs in memory or in persistent stores. Jena1 provided an additional API for manipulating DAML+OIL [32].

User feedback on Jena1 suggested better integration between the DAML+OIL support and the RDF support to permit, for example, the storing of DAML models within databases. It also had proved too difficult to add further implementations of the rich *Model* API to Jena1.

In response to these issues, Jena2 has a more decoupled architecture than Jena1. The two key architectural goals of Jena2 are:

- Multiple, flexible presentations of RDF graphs to the application programmer. This allows graph data to be accessed and manipulated through higher-level interfaces.
- A simple minimalist view of the RDF graph to the system programmer wishing to manipulate data as triples. This is particularly useful for RDFS and OWL reasoning.

The first is layered on top of the second: any triple source can back any presentation API. Both the architectural goals provide extension points for system programmers. The presentation layer is the basis of both the existing *Model* API and the new *Ontology* APIs for OWL [12], DAML+OIL [32] and RDFS. The graph layer allows the development of new triple sources, both materialized triples, for example from database or in-memory triple stores, and virtual triples generated dynamically as a result of some processing, such as inference or access to legacy data sources. Jena2 provides inference support for both the RDF semantics [16] and the OWL semantics [26].

Jena supports a Semantic Web query language, RDQL [25], that can be used either on top of materialized graphs, or on the virtual results of RDFS or OWL reasoning. Complete queries can be passed into the underlying graph layers, so database-backed graphs can take advantage of SQL optimization.

A third presentation interface, the RDF WebAPI [31], provides web clients with query-based access to RDF graphs. This query-based access is also available at both the system and application programmer interfaces, and acts as a further unifying theme of the architecture.

1.2 The Semantic Web Standards

The RDF abstract syntax [20] provides triples as a universal data structure. The vocabulary [8] from RDF and RDFS provide a core set of properties and classes to use with these triples. These triples can be serialized as RDF/XML [5]. The formal meaning [16] of these triples and of the vocabulary permits entailments to be drawn.

The RDF Schema vocabulary permits the definition of new classes and properties to be used in the graph. These are augmented by the Web Ontology Language (OWL) [12], which provides three levels: OWL Lite (the weakest), OWL DL, and

OWL Full (the strongest). OWL Full is a semantic extension of RDF; Jena2's ontology support is targeted at OWL Full.
Future Semantic Web standardization is likely to include work on query languages, and possibly Web APIs for the Semantic Web.

2. JENA2 ARCHITECTURE OVERVIEW

The heart of the Jena2 architecture is the RDF graph, a set of triples of nodes. This is shown in the *Graph* layer (see figure 1). This layer, following the RDF abstract syntax, is minimal by design: wherever possible functionality is done in other layers. This permits a range of implementations of this layer such as in-memory or persistence triple stores.
The *EnhGraph* layer is the extension point on which to build APIs: within Jena2 the functionality offered by the *EnhGraph* layer is used to implement the Jena1 *Model*[1] API and the new *Ontology* functionality for OWL and RDFS, upgrading the Jena1 DAML API.
I/O is done in the *Model* layer, essentially for historical reasons.
The Jena2 architecture supports *fast path query* that goes all the way through the layers from RDQL at the top right through to an SQL database at the bottom, allowing user queries to be optimized by the SQL query optimizer.
We give some more detail on the three layers below.

2.1 The Graph Layer: Triples as the Universal Data Structure

The *Graph* layer is based on the RDF Abstract Syntax [20]. It is straightforward to implement any of:

- triple stores, both in memory and backed by persistent storage;
- read-only views of non-triple data as triples, such as data read from a computer file system hierarchy, or scraped from a web page;
- virtual triples corresponding to the results of inference processes over some further set of triples as premises.

Implementations of the *Graph* layer provided with Jena2 give a variety of concrete (materialized) triple stores, and built-in inference for RDFS and a subset of OWL.

2.2 The Model Layer: Views for Application Programmers

Jena2 maintains the *Model* API from Jena1 as the primary abstraction of the RDF graph used by the application programmer. This gives a much richer set of methods for operating on both the graph itself (the *Model* interface) and the nodes within the graph (the *Resource* interface and its subclasses).
Further, the DAML API is updated and enhanced in Jena2 to form an Ontology API that can be realized as a DAML API or an OWL API.

2.3 The *EnhGraph* Layer: Multiple Simultaneous Views

Both the *Model* and the *Ontology* layers lie on top of the *Graph* layer via an intermediate layer: the *EnhGraph* layer.
This provides an extension point for providing views of graphs, and views of nodes within a graph. This generalizes the needs of both the *Model* and the *Ontology* APIs, and, significantly, makes the design decision that such presentation layers must be *stateless*: all significant state is within the graph. (Caching of state is permitted by the presentation layers). The *EnhGraph* layer is designed to permit multiple views of graphs and nodes which can be used simultaneously. Java's single inheritance model is sidestepped to provide polymorphic objects within the *EnhGraph* layer. This allows the multiple inheritance and typing of RDFS to be reflected in Java.

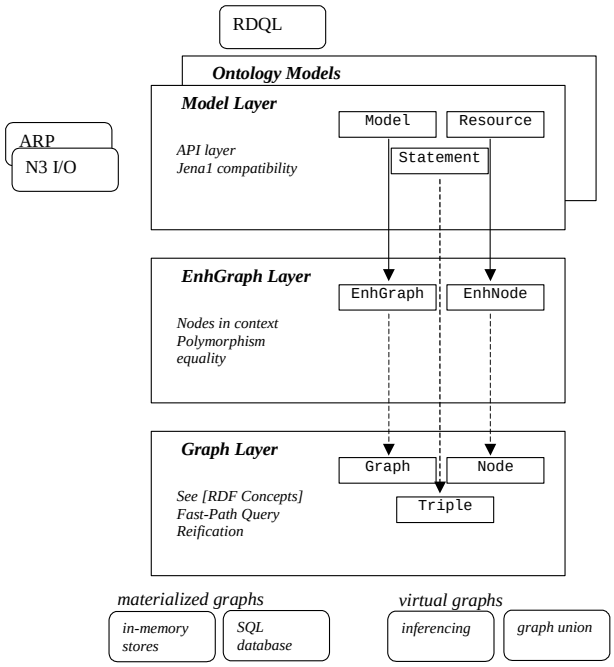

Figure. 1. The Jena2 Architecture

3. THE GRAPH SPI

Jena's implementation of RDF's abstract syntax [20] is the *Graph* SPI (System Programmer Interface), through which the triples of all Jena graphs are accessed.

3.1 Overview

The *Graph* layer defines an interface representing RDF graphs. The design goals for the *Graph* layer include:

- allowing collections of triples to be queried and updated efficiently. In particular, querying triples held in databases should be able to exploit the underlying database engine.
- being easy to reimplement, so that new triple collections can be represented by *Graph*s with minimal programming effort.
- supporting some specialist operations from the *Model* API when these cannot be easily constructed from the base functionality, reification in particular.

The elements within a `Graph` are `Triples`; each `Triple` comprises three `Nodes`, the subject, predicate, and object fields. A `Node` represents the RDF notion of a URI label, a blank node[2], or a literal; there are also two variable nodes, for named variables and a match-anything wildcard, for use in the `Query` interface. The RDF restrictions that a literal can only appear as an object, and that a property can only be labelled with a URI, are not enforced by the *Graph* layer but by the *Model* layer.
The core `Graph` interface supports modification (add and delete triples) and access (test if a triple is present or list all triples present matching some pattern). `Graph` implementations are free to restrict the particular triples they regard as legal and to restrict

[1] The term "Model" is taken from the original RDF Model & Syntax Recommendation [16], meaning data model.

[2] Also known as an anonymous resource, or *bNode*.

mutability – this is reflected through the use of Java exceptions and testable through a `Capabilities` interface.

The most significant part of the core `Graph` interface is the `find` operation. The primitive `find(Node S,Node P,Node O)` delivers an *iterator* over all the triples of the `Graph` which "match" the triple `(S, P, O)`. To "match" means to be equal to or for the S/P/O node to be ANY. This allows the `Graph` to be queried for e.g. all the properties of some particular subject, all the predicates with some particular object, or indeed all the triples in the `Graph`. This is the extensibility point that the inference engines and `Graph` combinators use for generating virtualized triples.

3.2 Fast Path Query

One of the goals of the *Graph* layer is to allow queries to be expressed which can exploit underlying efficient query engines, and which can return different kinds of results - variable bindings or subgraphs, for example.

Rather than add many operations to `Graph` itself, each `Graph` has an associated *query handler* which manages the more complex queries. A standard simple query handler is provided which implements the complex queries in terms of the `find` primitive for `Graph`s not offering more efficient possibilities.

A `Query` consists of a collection of triple patterns to be matched against some `Graph`s. A triple pattern is a `Triple` that may contain the extended ANY and Variable nodes mentioned above. So a `Query` might contain

```
(?x P ?y) (?y Q ?z)
```

to request all the bindings for ?x, ?y, and ?z for which matching triples can be found in the `Graph`. The query is executed so as to find all possible bindings of the variables; from this matched subgraphs can be computed.

Jena's memory-based `Graph` model simply implements the triple pattern matches by iterating over the `Graph` using `find`. The RDB-based `Graph`s instead compile some queries into SQL to be submitted to the database query engine.

The query handling operates over all the triples expressed by the `Graph`, however they are generated - as base assertions or as inferred consequences.

RDQL [25] uses this interface to do the non-constraint parts of its query handling.

3.3 Reification

RDF Model & Syntax [21] suggests making statements about statements by means of *reification*, representing the original triple as four triples forming what is known as a reification quad.

The new RDF recommendations reveal that the reification syntax does not completely achieve this goal, but continue to recommend the use of reification for provenance. This is in keeping with the practice of those Jena users who use reification heavily, to be able to add metadata to triples.

An important optimization is to be able to treat reification quads efficiently. Jena provides an API notion of a `ReifiedStatement` that encodes a statement in a model as a reification quad. This API-layer notion is reflected down into the `Graph` interface. Each `Graph` has an associated `Reifier` that is responsible for storing reified `Triple`s compactly. (This separation into a separate interface and implementation keeps the `Graph` interface uncluttered and allows different `Graph` implementations to share code.) `ReifiedStatement`s may be created by a single API call; thus users who explicitly reify many Statements need not pay a large cost in reification quads.

3.4 Other Details of the Graph Layer

Datatypes. The requirements of RDF manipulation of typed literals [16] differ from the Java norms. The same semantic value can have multiple RDF representations (e.g. `1^^xsd:int` and `1^^xsd:short`). One is initially tempted to arrange the Java equality operation on `Literal`s to take this value mapping into account. However, this interferes with the use of `Literal`s in indexed collections. Instead we introduced a separate notion of `sameValueAs` and arrange that searches on `Graph`s default to using this notion of equivalence in preference to Java equality.

Size. `Graph`s are sets of triples; a set naturally has a size (possibly infinite). However, with the notion of inferencing `Graph`s, implementing size is tricky and its behaviour non-obvious. While Jena `Graph`s have a `size` method, its meaning is inexact; for inference graphs it means "at least this many", not "exactly this many".

Transactions. Jena `Graph`s may support *transactions* such that changes to a `Graph` may be committed or abandoned. At present only RDB-based models offer this capability.

Bulk Update. If needed, triples may be added to or removed *en masse* rather than one at a time for efficiency reasons, e.g. for initialising a database.

4. ADDING APIS

Within the *Graph* layer, it is easy to provide triples. However, it is not easy to work with them at the application level (see [22]). Thus Jena includes the *Model* API to act as a presentation layer above the raw *Graph*, matching the abstractions of the RDF Vocabulary [8].

A key abstraction in the *Model* API is the `Resource`, corresponding to **`rdfs:Resource`**. This is a URIref or blank node seen as part of a particular *Model* - not merely a node but the collection of facts about that node in a particular RDF graph. From the point of view of the recommendations, a `Resource` provides a view not of the node, which is merely a URIref, but of its interpretation within an RDF graph.[3]

Sometimes it is useful at the application level to view and manipulate a `Resource` with a richer set of primitives based on extended functionality reflecting some aspect of its interpretation in a particular graph. For example, if a URIref is known to have type **`rdf:Bag`**, then viewing the corresponding `Resource` as a `Bag` allows easier access to its members without having to explicit access the **`rdf:_1`**, **`rdf:_2`** … triples. In this way we find direct support for RDF containers, as defined by the RDF Vocabulary, within the *Model* layer. Similar API classes and methods are provided to support the RDF reification vocabulary, which is implemented using the reification primitives in the graph layer (see section 3.3).

The *Model* layer is shown in figure 1, along with an alternative presentation layer: the *Ontology* layer. In the *Ontology* layer we find explicit support for concepts from RDF Schema and OWL. *Ontology* support adds many additional views of a URIref node within a particular graph, e.g. as an `OntClass` or a `CardinalityRestriction`. The functionality of the *Model* layer has previously been published, for example [22]; the *Ontology* layer is described below (section 9).

[3] The interpretation according to the RDFS semantics required is only reflected when an RDFS reasoner is used, see section 5.

4.1 The Enhanced Graph Layer

Jena2 does not attempt to present one consolidated presentation API onto an RDF graph. A consolidated API might be one in which a URIref with `rdf:type` of `owl:ObjectProperty` would be realized as a Java object of class `ObjectProperty`, which would implement all the interfaces that were appropriate, and inherit from some appropriate superclass. There is no obviously correct class hierarchy for the different concepts within the *Model* API and *Ontology* API. Moreover choosing which view to take of a particular URIref in the graph tends to make a closed world assumption: this resource **is** an RDFS class; later information may cause us to reconsider. A direct mapping of the class hierarchy of RDF onto that of Java would thus be a mistake.

To address this issue a framework for using many different presentation layers is provided, with two built-in instantiations, the *Model* API and the *Ontology Model* API.

4.1.1 Presentation Layers and Personalities

Each presentation layer consists of some interfaces and some implementation classes, and a mapping from interfaces to factory methods that invoke the implementation classes. This mapping is known as the `Personality` of the presentation layer. These presentation layers are stateless. All state is stored in `Graph`s.

The implementation classes either extend `EnhGraph` or `EnhNode`. `EnhGraph` is a simple wrapper around a `Graph`, with a pointer to the `Personality`. `EnhNode` is a wrapper around a `Node`, with an additional field pointing to an `EnhGraph`. This is the context in which the `EnhNode` is seen. For example, `Model` extends `EnhGraph`, and `Resource` extends `EnhNode`.

Each subclass of `EnhGraph` provides a set of operations that can be performed on a `Graph`. Thus an instance acts as a view of a `Graph`.

Each subclass of `EnhNode` provides a set of operations that can be performed on a `Node` in a `Graph`. Thus an instance acts as a view of a `Node` in a specific `Graph`. It is often appropriate to take different views of the same `Node`.

4.1.2 Polymorphism

RDFS permits any resource to have multiple types, which may or may not be related. `rdfs:subClassOf` can be used to express multiple inheritance and not just a simple hierarchy. In contrast, Java objects have a single Java class, from a tree of classes with single inheritance. Jena2 implements polymorphic resources to model within Java the RDFS style of multiple typing.

Given a `Node` or an `EnhNode` it is always possible to use the factory methods in the `Personality` to create a view (an instance of a subclass of `EnhNode`) of the `Node` that implements a particular interface. The constraint that these `EnhNode`s are stateless ensures that it is always safe to create more than one identical `EnhNode`. It also permits caching of `EnhNode`s for performance reasons. Thus a key method on an `EnhNode` is `as(interface)`, which returns a view of that `Node` which implements the given interface. If the desired result is not in a simple cache, then the view is created using the factory methods found in the `Personality`.

This method `as` is used while implementing a presentation API such as the *Model* or the *Ontology* layer as part of an extended casting idiom e.g.

```
(OntClass) ontProp.as(OntClass.class);
```

This is particularly useful for implementing OWL Full [27], in which there is no separation of vocabulary.

4.2 User defined Presentation APIs

Applications may well want to have their own specialized views of `Resource`s. Within the Jena2 architecture these views can be generated from an RDF Schema or an OWL Ontology and incorporated within the overall framework. That is, while the figure shows two instantiations of the top layer, other instantiations can be added.

5. INPUT/OUTPUT

The *de facto* reference parser for RDF/XML [5] is part of Jena. This is used at the W3C RDF validator site.

Jena I/O is provided at the *Model* layer, for historical rather than technical reasons. The basic primitives are to read and write models, in a choice of Semantic Web languages; Jena supports RDF/XML [5], N3 [7] and N-Triple [13].

5.1 Parsing RDF/XML

The RDF/XML parser uses Xerces to parse the XML, and validates the RDF input against a range of standards incorporated by reference into RDF/XML. Specifically, input is checked for conformance with RFC 2396 (URIs), RFC 3066 (languages, including ISO 601 and ISO 3166 checking), and the W3C Character Model.

The parser architecture, described in [9], cleanly separates the RDF processing and the XML processing, which has permitted the parser to closely track the revised RDF/XML working drafts as they evolved to the new Recommendation.

In Jena2, the implementation of that architecture has been improved, using a single Java thread, rather than two threads (for the XML parser and the RDF parser) as in Jena1.

5.2 Unparsing RDF/XML

RDF/XML output can be performed either in a basic mode, which is preferred for large files, or in a pretty writer mode. The former groups the triples of the graph by subject and then writes each triple using a simple property element - the abbreviations provided in RDF/XML are not used.

The pretty writer format, intended for when the output may be seen by people, is significantly more expensive, and uses all the productions in the revised syntax. *Striping*, as described in the recommendation [5], is the default. One of the options to control the output switches off rules as specified in the revised syntax. This option takes as an argument the URL references of the productions which are not to be used: e.g.

```
RDFWriter w = m.getWriter("RDF/XML-ABBREV");
String rdfSyntax =
    "http://www.w3.org/TR/rdf-syntax-grammar";
w.setProperty("blockRules",
    rdfSyntax+"#idAttr,"+
    rdfSyntax+"#parseTypeCollectionPropertyElt");
```

Since, as described in [10], the code closely follows the formal grammar in the W3C recommendation, the URL references used refer directly to the grammar, making it easy for users to understand how to use the controls.

Feedback from users has indicated that the property attribute rule in RDF Syntax is poorly understood, and we changed the default to not use it. Further feedback has indicated that users value the preservation of the prefixes used for XML namespaces. Changes to both the RDF/XML input and output have implemented this functionality.

6. INFERENCE SUPPORT

The treatment of schema and ontologies in the formal semantics of both RDF [16] and OWL [26] makes it clear that entailment is a

core feature of the Semantic Web recommendations. This is also reflected in the hundreds of entailments tests provided [11], [13].
Jena supports this by giving access to a range of inference capabilities. A core set of such capabilities are available "out of the box", particularly RDFS inference, and OWL inference supporting the subset of OWL Full roughly corresponding to the union of OWL Lite and RDFS. The architecture permits plug-in connections to engines being developed by the wider community, such as Racer [15], FaCT [18] and the Java Theorem Prover [14]. It is planned that using such plug-ins complete OWL Lite and improved OWL DL reasoning will be supported.
The design center for Jena2's inference API is to enable applications to access RDF data that has been enriched by additional assertions entailed from a set of relevant ontologies – that is we emphasize Abox queries over Tbox queries [2]. This bias influences our choice of API, architecture and inference engines.

6.1 Inference access API and architecture

In Jena2, inference engines are structured as `Graph` combinators called `Reasoner`s. An instance of a `Reasoner` combines one or more RDF Graphs and exposes the entailments from them as another RDF Graph in which some of the retrievable triples are virtual entailments rather than materialized data (see figure 2). The input `Graph`s contain both the ontology and instance data, with optional separation between the two. In particular, it is possible to partially-bind a `Reasoner` to an ontology and then use the resulting specialized `Reasoner` to access multiple different data `Graph`s – reusing the ontology inferences.
This layering offers great flexibility. For example, an OWL Lite `Reasoner` can be stacked on top of an RDFS `Reasoner` with the latter being used to infer the type statements required by the OWL Lite syntax but deducible from the domain/range declarations of the OWL Lite properties. An RDQL query can be issued to an inferred `Graph`, just like any other `Graph`, thus allowing query over the entailments. For instance an inferred RDFS graph can be used by RDQL to do schema directed queries, and hence have functionality more like RQL [19], without any additional query syntax, and with better alignment with the RDF recommendation.
Different APIs can be bound to the inferred `Graph` allowing the results to be viewed at the RDF level or though the convenience ontology API.

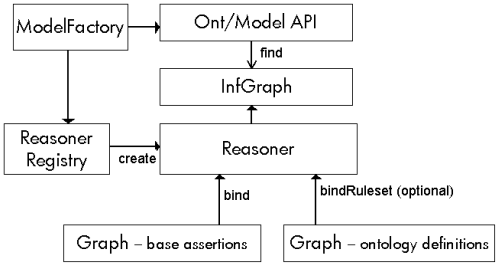

Figure 2. **The inference API layering**

Many entailments are easily accessible through the *Graph* or *Model* APIs in this way. For example finding all instances of a given class is a simple triple query:

```
(?x rdf:type C)
```

where `C` is a `Node` representing an `rdfs:Class` in the ontology `Graph` (and can be either a bnode or a named class).

There are several limitations of this API approach that we had to address.
First, arbitrary class expressions introduced by the comprehension axioms of OWL [27] are not directly supported. We addressed the same requirement by extending the `find` operation to take an optional parameter, for additional premises. This parameter is a `Graph` containing expressions whose `Node`s can be used within the query triples; the intended use is with expressions known to be valid from the comprehension axioms.
Second, for transitive relations it is often convenient to be able to query the direct (or minimal) as well as the transitively closed version of the relations. We handle this by introducing additional RDF properties to represent the direct version of any transitive property. This style of extension, introducing additional RDF properties to represent inferable relationships, makes the triple-based API very flexible.
Thirdly, to give convenient access to consistency information we added a `validate` method which returns a report containing a list of all warnings or errors identified within the `Graph`. This is more convenient than the property introduction technique, in this case, because validity reports may need to refer to statements or groups of statements and not simply to `Node`s.

6.2 Built-in reasoners

As part of the default distribution we include a selection of inference engines, which conform to this architecture.

Transitive Reasoner. This reasoner provides the transitive closure of the RDFS `subClassOf` and `subPropertyOf` relationships contained in the source graphs, dealing correctly with cases such as declaring sub-properties of the `rdfs:subClassOf` property. This relatively simple functionality corresponds quite closely to the hard coded inference implemented by the Jena1 DAML+OIL API, enabling applications to make similar queries in Jena2 with similar performance without needing to invoke more sophisticated inference engines.

RDFS Reasoner. This provides an implementation of the RDFS closure rules [16]. It strikes a balance between eager and lazy processing. The sub-class and sub-property lattices are cached using an embedded `TransitiveReasoner`. Each domain, range, sub-property and sub-class declaration is eagerly translated into a single query rewrite rule. The result of a query to the graph will be the union of the results from applying the query plus all the rewritten versions of the query to the underlying graphs.

Rubrik[4] Reasoner. This reasoner supports rule based RDF inferences. Rule clauses are either extended triple patterns or procedural callouts to primitives defined in Java. The triple patterns are *extended* in the sense that the objects of the triple can be functor-like data structures. This allows rule authors to control the combinatorics of graph pattern matching by having one rule map a subgraph pattern into compact data structure and later rules fire off that data structure. Both forward chaining and memoized backward chaining rule engines are provided, with some hybridization in that forward rules are able to create and install new backward rules.

These rule engines may be used with application-specific rule sets or with prepackaged rule sets for RDFS and for the OWL Lite subset of OWL Full. The rule-based approach corresponds well to our emphasis on ABox reasoning: indeed, we handle class-

[4] RDF basic rule inference kit.

subsumption checking by introducing prototypical instances of classes and letting the instance rules determine the other type labels for the prototype.
In addition to these inference engines we have found it useful to provide a set of operators (union, multi-union, intersection, difference, delta) for combining data `Graph`s.
The possibility of custom rules within the rubrik reasoner has generated user interest, indicating that this approach complements more complete OWL reasoning. Other user feedback has resulted in (incomplete) support for owl:hasValue being added to the OWL rulebase.

6.3 External reasoners

By using such a generic, triple-based, interface onto inference results it becomes possible to expose the capability of a wide range of reasoning engines through the same API. We enhance this flexibility by providing a `ReasonerRegistry` in which available reasoners can be registered along with an identifying URI and a reasoner capability description (expressed in RDF). In this way applications can be made somewhat independent of the particular inference engine being used.
We intend to construct adaptors for several openly available reasoners, to enable their use from within Jena.

7. RDQL – RDF QUERY

RDQL (RDF Data Query Language) was pioneered in Jena1. The Jena implementation is the *de facto* reference implementation. A full description is found in [29], the original paper is [25].
An RDQL query consists of a graph pattern, expressed as a list of triple patterns. Each triple pattern is comprised of named variables and RDF values (URIs and literals). An RDQL query can additionally have a set of constraints on the values of those variables, and a list of the variables required in the answer set.

```
SELECT ?x
WHERE (?x,
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>,
<http://example.com/someType>)
```

This triple pattern matches all statements in the graph that have predicate `http://www.w3.org/1999/02/22-rdf-syntax-ns#type` and object `http://example.com/someType`. The variable "?x" will be bound to the label of the subject resource. All such "x" are returned
An RDQL query treats an RDF graph purely as data. If the implementation of that graph provides inferencing to appear as "virtual triples" (i.e. triples that appear in the graph but are not in the ground facts) then an RDQL will include those triples as possible matches in triple patterns. RDQL makes no distinction between inferred triples and ground triples.
The next phase of the Semantic Web activity by the W3C is likely to address RDF query. We hope that this work will take our positive experinces with RDQL into account. Jena's RDQL implementation will evolve as a result of the new work at the W3C.

8. PERSISTENT STORAGE

As in Jena1, the database subsystem in Jena2 supports persistent storage of RDF Models in a conventional database [33]. Implemented at the Graph layer, it provides all the usual Graph operations (add, delete, find) and efficiently supports reification.

8.1 Denormalized Schema

Jena2 stores each triple either in a general purpose triple table or a property table, for a specific property.
The interface trades-off space for time. It uses a *denormalized* schema in which resource URIs and simple literal values are stored directly in the triple table. A separate literals table is used only to store literal values whose length exceeds a threshold or that are typed or have a language tag. This makes it possible to process a large class of queries without a join. However, a denormalized schema uses more database space because the same value (literal or URI) is stored repeatedly.
The increase in database space consumption is addressed in several ways. First, common prefixes in URIs, such as namespaces, are stored in a separate table and the prefix in the URI is replaced by a reference. This prefix table will be cacheable in memory so expanding a prefix does not require a database join.
Second, a literals table is used so that long literals are stored only once. Third, Jena2 supports *property* tables as described below. Property tables offer a modest reduction in space consumption in that the property URI is not stored.

8.2 Configuration

Configuration parameters are specified as RDF statements in a memory model that is passed as an argument when creating a new persistent model. Jena2 includes default models containing the default configuration parameters for all supported databases. Specifying configurations in RDF makes configurations easy to search, share and re-use since they can be manipulated with Jena's existing operations.

8.3 Property Tables

A property table (also known as attribute tables [1]) holds statements for a specific property. They are stored as subject-value pairs in a separate table. Triple tables and property tables are disjoint - a statement is only stored once. For properties with a maximum cardinality of one, it is possible to cluster multiple properties together in a single table. A single row of the table stores property values for a common subject. For example, a Dublin Core [6] property table might store `dc:title`, `dc:publisher`, `dc:description`.
Multi-valued properties, e.g., `dc:creator`, cannot be clustered and must be stored in separate tables. Note that if the datatype of the object value is known, it may be possible[5] to make the underlying database column for the value match the property type.
A *property class* table is a special kind of property table that stores properties associated with a particular class and also records all instances of that class. Each property must have the class as its domain. Jena2 implements reification as a property class table. The properties are `rdf:subject`, `rdf:predicate`, `rdf:object` and the class is constrained to be `rdf:Statement`. The subject of the property class table is the URI that reifies the statement.

8.4 Query Processing

Queries are executed against graphs which may have multiple statement tables. For each statement table there is a handler to convert between the graph view of Jena and the tuple view of SQL. To evaluate a triple pattern, the query processor passes the pattern, in turn, to each table handler for evaluation.
A goal of Jena2 is support for *fast path* query processing for RDQL (see section 3.2). In Jena1, an RDQL query was converted into a pipeline of triple pattern queries. This is evaluated in a nested-loops fashion in Java by using the results of one triple pattern to bind values to variables and then generating new triple

[5] Not all XSD types correspond to an SQL datatype.

patterns for evaluation. Jena2's RDQL uses the *Graph* query interface to pass all the triple patterns into the database graph; the goal of fast path query processing is to use the database engine to process the entire query, rather than single patterns.

A full discussion of fast path query processing is beyond the scope of the paper. Here, we present two simple cases and mention the difficulties for the general case. For the first simple case, assume that all the triple patterns reference only the triple table. As mentioned above, a single triple pattern can be completely evaluated over a table by a single SQL query. To evaluate multiple patterns in the database engine, it is sufficient to combine the SQL statements for the individual patterns and add additional join conditions for the linking variables.

The second simple case is when all patterns can be completely evaluated by a single property table. This is similar to the first case. However, here it may be possible to eliminate joins if the patterns reference properties stored together (since the property values for the same subject are stored in the same row).

When the triple patterns for a query apply to multiple tables, it is more difficult to construct a single SQL query to satisfy all patterns. The Jena1 nested-loops approach is applied in this case. We are currently investigating optimized solutions for the general case.

9. ONTOLOGY SUPPORT

Since Jena is, at heart, an RDF platform, we restrict ourselves to ontology formalisms built on top of RDF. Specifically this means RDFS [8], the varieties of OWL [12] and DAML+OIL [32].

While OWL builds on top of the RDF specifications, it is possible to treat OWL as a separate language in its own right, and not something that is built on an RDF foundation; see for example the OWL API [3], which merely uses RDF as a serialisation syntax. The RDF-centric view treats RDF triples as the core of the OWL formalism. While both views are valid, in Jena we take the RDF-centric view. As such, the ontology support within Jena addresses OWL Full features that are not present in OWL DL: e.g. the ability to use a single URIref to denote a class, a property and a participant in some other ontological schema.

The *Ontology* layer defines the interface `OntModel` which extends the Model interface from the *Model* API.

Rather than having Java class names that are tightly bound to the language being processed (e.g. `DAMLClass`, `DAMLObjectProperty`, etc.), the ontology API is language-neutral (thus the classes are `OntClass` and `ObjectProperty`). To support this, each of the languages has a *profile*, which lists the permitted constructs and the URI's of the classes and properties. Thus in the DAML profile, the URI for object property is `daml:ObjectProperty`, in the OWL profile is it `owl:ObjectProperty` and in the RDFS profile it is `null` since RDFS does not define object properties.

The profile is bound to an ontology model, which is an extended version of Jena's Model class. The general Model allows access to the statements in a collection of RDF data. OntModel extends this by adding support for the kinds of objects expected to be in an ontology: classes (in a class hierarchy), properties (in a property hierarchy) and individuals. The properties defined in the ontology language map to accessor methods. For example, an `OntClass` has a method to list its super-classes, which corresponds to the values of the `subClassOf` property. No information is stored in the `OntClass` object itself. When the OntClass `listSuperClasses()` method is called, the information is retrieved from the underlying RDF statements. Similarly adding a subclass to an `OntClass` asserts an additional RDF statement into the model.

The statements that the ontology Java objects see depend on both the asserted statements in the underlying RDF graph, and the statements that can be inferred by the reasoner being used (if any).

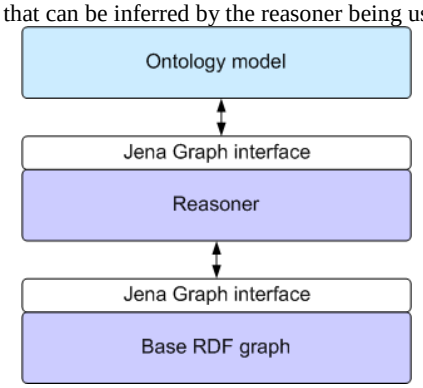

Figure 3. The statements seen by the OntModel

The asserted statements are held in the base graph. This presents the simple internal interface, `Graph`. The reasoner, or inference engine, can use the contents of the base graph and the semantic rules of the language, to show a more complete set of statements - i.e. including those that are *entailed* by the base assertions. This is also presented via the `Graph` interface, so the model works only with that interface. This allows us to build models with no reasoner, or with one of a variety of different reasoners, without changing the ontology model. It also means that the base graph can be an in-memory store, a database-backed persistent store, or some other storage structure altogether (e.g. an LDAP directory) again without affecting the ontology model.

9.1 RDF-level polymorphism and Java

Consider the following RDF sample:

```
<rdfs:Class rdf:ID="DigitalCamera">
</rdfs:Class>
```

This declares that the resource with the (relative) URI `#DigitalCamera` is an ontology class. It might be appropriate to model declaration in Java with an instance of an `OntClass`. Now suppose we augment the class declaration with some more information:

```
<rdfs:Class rdf:ID="DigitalCamera">
  <rdf:type owl:Restriction />
</rdfs:Class>
```

Now we are saying that `#DigitalCamera` is an OWL `Restriction` (which is a subclass of `rdfs:Class`). A problem we have is that Java does not allow us to dynamically change the Java class of the object modeling this resource. The resource has not changed: it still has URI `#DigitalCamera`. But the appropriate Java class we might choose to model it has changed from `OntClass` to `Restriction`. Conversely, if we remove the `rdf:type Restriction` from the model, the use of a `Restriction` Java class is no longer appropriate.

Even worse, OWL Full allows us the following (rather counterintuitive) construction:

```
<rdfs:Class rdf:ID="DigitalCamera">
  <rdf:type owl:ObjectProperty />
</rdfs:Class>
```

That is, `#DigitalCamera` is now a class *and* a property. While this may not be a very useful operation, it illustrates a basic point that we cannot rely on a consistent or unique mapping between an RDF resource and the appropriate Java abstraction.

Jena 2 accepts this basic characteristic of polymorphism at the RDF level by considering that the Java abstraction (`OntClass`, `Restriction`, `DatatypeProperty`, etc.) is just a view or *facet* of the resource. Given a RDF object in Jena, we can get a new facet with the `as()` method. For example:

```
Resource r = myModel.getResource(
                  myNS + "DigitalCamera" );
OntClass cls = (OntClass) r.as(
                   OntClass.class );
Restriction rest = (Restriction) cls.as(
                   Restriction.class );
```

This pattern allows us to defer until run-time decisions about the correct Java abstraction to use, and make this choice depend on the properties of the resource itself. If a given Resource will not support the conversion to a given facet, it will raise an exception.
This RDF-level polymorphism is used extensively in the Jena ontology API to allow maximum flexibility in handling ontology data.

9.2 Imports

The imports mechanism of OWL and DAML+OIL are usually handled as shown in Figure 4.
Figure 4 shows that each imported ontology document is held in a separate graph structure. If we did not do this, once the imports had been processed it would be impossible to know where a statement came from. It is possible to switch off imports processing.

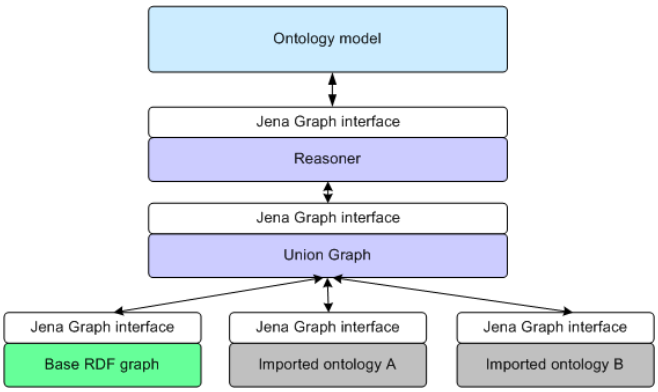

Figure 4. Ontology internal structure including imports

9.3 OntResource

All of the classes in the ontology API that represent ontology values have `OntResource` as a common super-class. `OntResource` contains shared functionality for the ontology classes. Example properties are annotations, such as `label`, and `sameAs`.
For each of these properties there is a standard pattern of available methods, to set and get a singleton value or modify or inspect a multivalue.
`OntResource` also provides methods for listing, getting and setting the RDF types of a resource. The `rdf:type` property is one for which many entailment rules are defined in the semantic models of the various ontology languages. Therefore, the values that `listRDFTypes()` returns is more than usually dependent on the actual reasoner bound to the ontology model. For example, suppose we have class A, class B which is a subclass of A, and resource x whose asserted `rdf:type` is B. With no reasoner, listing x's RDF types will return only B. If the reasoner is able to calculate the closure of the subclass hierarchy (and most can), X's RDF types would also include A. Furthermore, complete reasoners might also infer that x has type `owl:Thing` and `rdf:Resource`.
For some tasks, getting a complete list of the RDF types of a resource is exactly what is needed. For other tasks, this is not the case. An ontology editor, for example, might want to distinguish in its display between inferred and asserted types. In the above example, only `x rdf:type B` is asserted, everything else is inferred. One way to make this distinction is to make use of the base model or of the union graph containing the base model and the imported graphs (see Figure 4). Getting the x resource from the base model and listing the type properties there would return only the asserted values.

9.4 Other Aspects

As one might expect, the Ontology API contains classes corresponding to the main concepts within OWL, and the ability to inspect and manipulate the principal properties of these classes. The same API can be used for DAML+OIL, and a subset of the API is useful for RDFS.
Moreover, the base model can be stored in a database, if required.
A further independent module is an OWL Syntax Checker, conforming with the specification given in the OWL Test Cases [11]. It inspects a set of triples and determines whether they fit within the OWL Lite or OWL DL or OWL Full syntactic species of OWL. It's triple oriented operation is described in [4].

10. VIEWS IN JENA2

We have seen that the *Graph* interface acts as a uniform interface into triples, both actual, as found in documents or databases, or virtual, as defined by arbitrary Java code, in particular reasoners. Similar but more constrained mechanism are found in TRIPLE [24] and are proposed as RVL [23]. In each of these, views of virtual triples are defined using a high-level view definition language in terms of other views or collections of materialized triples.

It would be possible to translate such high-level view definition languages into Java code, and, with care, features of Jena2 such as fast-path query could be utilized. Moreover, the Jena2 architecture shows a continuum between such view languages and inference. The rule language in Jena2 can be seen in itself as a view definition language, although with a somewhat different intent from RVL or TRIPLE views. In particular, the reasoners in Jena *add* triples, whereas an RVL or TRIPLE view both adds new triples and hides the old triples.

11. JOSEKI

A further feature of Jena that we hope will be standardized in the next round of the Semantic Web activity is that of a WebAPI [30].

Joseki takes the RDF graph as the primary design concept and makes it accessible to remote clients and applications. The RDF WebAPI provides a simple, universal access mechanism for an application on one machine to extract information from an RDF repository hosted by another machine. The access mechanism is graph-based query where the access to the remote knowledge base is a query and the results are expressed in terms of a single graph.

11.1 Query as access primitive

The WebAPI fits into the Jena paradigm by providing access to a graph. This graph may be ground data or it may be a graph where inference is performed to yield entailed triples. This is not visible to the remote client application. The contract between information consumer and the information publisher is that the graph is a set of triples expressing some information. How this

information is derived is purely a matter for the information provider and not part of the contract between provider (publisher) and the client application.

Query forms the access paradigm because it is not desirable to copy such knowledge bases across networks, This may be because they are large, and the client application is only interested in a small part of the overall graph, or because the knowledge base is frequently changing, making locally caching ineffective.

A query returns a single subgraph. Unless the query further modifies the request with additional parameters, the contract is that the subgraph returned should yield the same matching results as the original query would on the entire knowledge base. The minimal complete subgraph is the smallest such graph. In a conjunctive query language such as RDQL [25], this is equivalent to calculating the result triples by substituting each of the query solutions into the graph pattern and merging into a single graph.

The graph returned does not have to be minimal. A sophisticated cache could return a precomputed, or previous computed, subgraph that is larger than the minimal matching subgraph but still meets the completeness requirement.

Query provides a sufficiently coarse-grained operation for efficient application use. Direct triple access would cause large numbers of fine-grained network accesses leading to excessive overhead.

11.2 The RDF WebAPI

To make RDF repositories available across the Internet, the RDF WebAPI requires each graph to have a URL for the purposes of naming and routing query traffic to the repository providing that graph. One host repository may have several RDF graphs available, so it is necessary to direct queries to the right one based on both network location and on name. URLs provide the mechanism for this.

The protocol used for query is HTTP, specifically the GET verb. In order to provide compatibility with regular web use, a plain GET (no query string provided) is interpreted as fetching the whole RDF graph. A query string provides refinement of the GET to extract a subgraph of the target graph. The query string consists of identification of the query language and a query-language specific string giving the query itself.

The full details of this can be found in the member submission to the W3C [30].

11.3 Joseki – Client Library

The client library provides integration with the rest of the Jena2 API in two ways. First, the primitive operation of querying and returning the minimal, complete subgraph is provided where the remote query processor is expected to compute a complete subgraph. Second, a remote query engine, matching the standard QueryEngine interface in the *Graph* layer is also provided. This latter access mechanism yields an iterator of bound variables just like a local query. The variable bindings are locally calculated based on the complete subgraph returned by the remote operation.

12. CONCLUSION

Jena2 provides integrated implementations of the W3C Semantic Web Recommendations, centred on the RDF graph. Moreover, additional features of Jena: the query language and the WebAPI, are ones that we hope will be finalized in the next phase of the Semantic Web activity.

The Jena2 architecture cleanly separates presentational issues, concerning what the application programmer wishes to do with an RDF graph, from the system programming issues such as how to store concrete triples or derive virtual triples. This enables the following new features in Jena2:

- RDFS inference support, following the RDF Semantics.
- Full integration of the Ontology support with other Jena components. The Ontology presentation API can be layered, with or without any inference support, over any triple store.
- RDQL can be used to query the virtual triples resulting from RDFS or OWL inferences.
- Adding a new extensibility point in Jena for integrating DL reasoners such as Racer and FACT, as part of improving the ontology support and, in particular, support of OWL.
- Integration of query optimizers, for example, by passing back-end relational databases complex SQL queries representing the user level query, rather than merely using a relational database as a triple store.
- Seamless extension to access over the web.

Jena2 is available under a BSD-style license from `http://jena.sourceforge.net`.

13. ACKNOWLEDGEMENTS

Jena2 is the work of the whole Jena team: Harumi Kuno, Brian McBride, Craig Sayers and the authors. Thanks too, to other participants on the jena-devel mailing list, and to all the Jena users who have contributed bug reports or suggested enhancements.

14. REFERENCES

[1] S. Alexaki, V. Christophides, G. Karvounarakis, D. Plexousakis, K. Tolle, *The ICS-FORTH RDFSuite: Managing Voluminous RDF Description Bases*, 2nd Intl Workshop on the Semantic Web (SemWeb'01, with WWW10), pp. 1-13, Hongkong, May 1, 2001.

[2] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, P. Patel-Schneider *The Description Logic Handbook*, 2003, CUP.

[3] S.Bechhoffer, R. Volz, P. Lord. *Cooking the Semantic Web with the OWL API*, Proceedings of ISWC 2003, pp 659-675.

[4] S.Bechhoffer, J.J.Carroll, *Parsing OWL DL: Trees or Triples*, WWW2004 New York.

[5] Beckett D. *RDF/XML Syntax Revised*, 2004, W3C[6].

[6] Beckett, D., Miller E., Brickley, D., *Expressing Simple Dublin Core in RDF/XML, DCMI Recommendation*, 2002. http://dublincore.org/documents/2002/07/31/dcmes-xml/

[7] T. Berners-Lee et al. *Primer: Getting into RDF & Semantic Web using N3*, http://www.w3.org/2000/10/swap/Primer.html

[8] D.Brickley, R.V.Guha, *RDF Vocabulary Description Language 1.0: RDF Schema*, W3C[6].

[9] J.J. Carroll *CoParsing of RDF & XML*, HP Labs Technical Report, HPL-2001-292, 2001

[10] J.J. Carroll, *Unparsing RDF/XML*,WWW2002. http://www.hpl.hp.com/techreports/2001/HPL-2001-292.html

[11] J.J. Carroll, J. De Roo, *OWL Test Cases*, 2004, W3C[6].

[12] M. Dean, G. Schreiber, *OWL Reference*, 2004, W3C[6].

[13] J. Grant, D. Beckett, *RDF Test Cases*, 2004, W3C[6].

[14] Gleb Frank *A General Interface for Interaction of Special-Purpose Reasoners within a Modular Reasoning System*, in: "Question Answering Systems. Papers from the 1999 AAAI Fall Symposium," pp. 57-62.

[15] Volker Haarslev, Ralf Möller *Description of the RACER System and its Applications*, Intl Workshop on Description Logics (DL-2001), Stanford, USA, 1.-3. August 2001

[16] P. Hayes, *RDF Semantics*, 2004, W3C[6]

[17] I. Horrocks, J. Hendler (eds) *The Semantic Web - ISWC 2002*, First International Semantic Web Conference, Sardinia, Italy, June 9-12, 2002. Proceedings, Springer.

[18] I. Horrocks. "Using an expressive description logic: FaCT or fiction?" In A. G. Cohn, L. Schubert, and S. C. Shapiro, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Sixth International Conference (KR'98)*, pages 636-647. Morgan Kaufmann Publishers, San Francisco, California, June 1998.

[19] G. Karvounarakis, V. Christophides, D. Plexousakis, S Alexaki, *Querying Community Web Portals*, SIGMOD2000.

[20] G. Klyne, J.J. Carroll, *RDF Concepts and Abstract Syntax*, 2004, W3C[6].

[21] O. Lassila, R.R. Swick, *RDF Model & Syntax* 1999, W3C[6].

[22] B. McBride *Jena* IEEE Internet Computing, July/August, 2002.

[23] A. Magkanaraki, V. Tannen, V. Christophides, D. Plexousakis. *Viewing the Semantic Web through RVL Lenses*, Proc. of ISWC 2003 pp 96-112.

[24] Z. Miklós, G. Neumann, U. Zdun, M. Sintek *Querying Semantic Web Resources Using TRIPLE Views*, Proc. of ISWC 2003 pp 517-532.

[25] L. Miller, A. Seaborne, and A. Reggiori *Three Implementations of SquishQL, a Simple RDF Query Language***,** 2002, p 423 ff. in [17].

[26] P.F. Patel-Schneider, P. Hayes, I. Horrocks, *OWL Semantics & Abstract Syntax*, 2004, W3C[6].

[27] P.F. Patel-Schneider, P. Hayes, I. Horrocks, *OWL: RDF-Compatible Model-Theoretic Semantics* 2004, in [26]

[28] D. Reynolds, *Jena Relational Database Interface – Performance Notes*, in Jena 1.6.1 download: http://www.hpl.hp.com/semweb/download.htm

[29] A. Seaborne *RDQL— A Query Language for RDF*, 2003, http://www.w3.org/Submission/2004/SUBM-RDQL-20040109/.

[30] A. Seaborne *RDF Net API*, 2003, http://www.w3.org/Submission/2003/SUBM-rdf-netapi-20031002/

[31] A. Seaborne *An RDF NetAPI*, 2002, p. 399 ff in [17]

[32] F. van Harmelen, P. F. Patel-Schneider I. Horrocks, *Reference description of the DAML+OIL (March 2001) ontology markup language*, http://www.daml.org/2001/03/reference

[33] K. Wilkinson, C. Sayers, H. Kuno, D. Reynolds, *Efficient RDF Storage and Retrieval in Jena2*, HP Laboratories Technical Report HPL-2003-266

[6] W3C Technical Reports can be found at http://www.w3.org/TR/