A Semantic Web Resource Protocol: XPointer and HTTP

Kendall Clark[1], Bijan Parsia[2], Bryan Thompson[3], Bradley Bebee[4]

[1] University of Maryland, MIND Lab
8400 Baltimore Ave,
College Park MD 20742, USA
kendall@monkeyfist.com
[2] University of Maryland, MIND Lab
8400 Baltimore Ave,
College Park MD 20742, USA
bparsia@isr.umd.edu
[3] SAIC, Advanced Systems and Concepts
3811 N. Fairfax Dr., Suite 850
Arlington VA 22203, USA
BRYAN.B.THOMPSON@saic.com
[4] SAIC, Advanced Systems and Concepts
3811 N. Fairfax Dr., Suite 850
Arlington VA 22203, USA
bebeeb@saic.com

Abstract. Semantic Web resources — that is, knowledge representation formalisms existing in a distributed hypermedia system — require different addressing and processing models and capacities than the typical kinds of World Wide Web resources. We describe an approach to building a Semantic Web resource protocol — a scalable, extensible logical addressing scheme and transport protocol — by using and extending existing specifications and technologies. We introduce XPointer and some infrequently used, but useful features of HTTP/1.1, in order to support addressing and server side processing of resource and subresource operations. We consider applications of the XPointer Framework for use in the Semantic Web, particularly for RDF and OWL resources and subresources. We describe two initial implementations: filtering of RSS resources by date and item range; RDF subresource selection using RDQL. Finally, we describe possible application to the problem of OWL imports.

1 RDF, RDFS, and OWL

RDF, RDFS, and OWL form the middle layers of the so-called Semantic Web layer cake. They are also its transitional layers; the parts beneath RDF-RDFS-OWL are part of the existing Web and are not distinctively *semantic*. These lower layers — URIs, Unicode, XML, and W3C XML Schema (WXS) — are not KR formalisms, though they do contribute, especially XML and WXS, to the syntax and the semantics of RDF, RDFS, and OWL. RDF, RDFS, and OWL

are minimalist formalisms, defined primarily as logics independent of particular theories of the world or of the Web. At best they have a thin upper ontology for representing aspects of their own abstract syntax and semantics. For example, the hierarchy of builtin classes of OWL Full, which is the richest of the OWL variants in terms of expressiveness and predefined ontology, contains a universal class (`rdfs:Resource`), several classes of classes and of properties (`rdfs:Class`, `owl:Class`, `rdf:Property`, `rdf:AnnotationProperty`, etc.), along with a few individuals (`rdf:nil`).[5]

Further, certain kinds of inference are licensed in RDF, RDFS, and OWL. If, for example, an RDF graph contains a triple `(s p o)`, then there is at least one other triple `(p rdf:type rdf:Property)` that is RDF-entailed by that graph. This is unusual from a first order logic point of view; it is not merely a syntactically sensitive inference. Rather, it derives a claim *about* that syntax, which highlights the strong reflective modeling capabilities deemed essential to the Semantic Web. But putting aside, first, the desirability of such reflective modeling capabilities or, second, the specific capabilities of OWL Full, there are two ways in which the thin upper ontology formed by RDFS and OWL Full, together with the associated inference mechanisms, fails to cover existing, critical aspects of the Web and the Semantic Web:

1. There are no classes or properties to represent RDF graphs or documents themselves. RDF graphs are a fundamental kind of Semantic Web resource.
2. There is no connection between the defined semantics of URIs and the assertions, especially type assertions, using them. For example, it is not unreasonable, and it has sometimes been proposed, that `mailto:'` URIs only denote things that are mailboxes or mail address endpoints.

Since there appears to be no consensus for specific sets of URI as to their proper associated type, standardization of such inferences would be premature. However, it is important to be able easily to coin sensible URIs for mailboxes, Web pages, people, and RDF graphs. Taking just the case of RDF graphs, it is simple enough to offer some sensible principles to resource publishers and URI owners.[6] For example, treat RDF graphs as normal Web accessible information resources. The simplest application of this principle is to publish RDF graphs as RDF/XML documents on the Web, allowing others to interact with those graphs using HTTP.[7] This deployment practice will suffice if two conditions are

[5] Of course, in OWL Full all classes and properties are `rdfs:Resource` and, thus, individuals in the domain as well.

[6] The canonical discussion of URI ownership is [10], especially section 2.2.1.1.

[7] Or, as [10] puts it,

> A URI owner may, upon request, provide representations of the resource identified by the URI. For example, when a URI owner uses the HTTP protocol to provide those representations, the HTTP origin server...is the software agent acting on behalf of the URI owner to provide the authoritative representations for the resource identified by that URI. The owner is also responsible for accepting or rejecting requests to modify the resource identified by that URI...

met: first, that the graphs we want to manipulate directly are relatively small; and, second, that we can anticipate the subgraphs users will need and provide the appropriate document interface to them.

When RDF graphs or OWL ontologies are very large[8], retrieving the entire graph as a document to be processed locally by the client is often impractical, especially if the graph is dynamic or if the client is resource-constrained. Further, it is not at all clear that standard techniques for mapping resource representations to documents fits Semantic Web resource representations very well. While for many applications RDF data has an obvious document chunking — for example, individual Web resource metadata can be embedded or associated with the corresponding resource — it can be difficult to anticipate the range of kinds of queries a client may need to execute. And many common queries may cut across the obvious or common case RDF chunking. While it is natural to associate creator information with created resources, it is perfectly sensible to want to know all the creators for a site, all the resources created by the current page's creators, and so on. In such cases, having to aggregate the chunking RDF is worse than simply downloading a very large RDF/XML document.

These problems and constraints point to the need for a logical, extensible means of identifying and addressing Semantic Web resources, preferably in a way that is as consonant as possible with existing standards and deployed systems.

2 Web Addressing and Fragment Identifiers

According to [2], in order to authoritatively interpret a URI fragment identifier,[9] a user-agent must dereference the URI containing the fragment identifier, using the Internet Media Type (IMT) [7] of the retrieved representation to apply an authoritative interpretation function[10] to the fragment identifier.

When dereferencing URIs that return HTML representations, the user-agent is usually acting on behalf of a human person who is navigating web hypermedia resources. In this case the semantics of the fragment identifier are interpreted by a web browser such that the subresource identified by the fragment identifier

---

[8] For example, the National Cancer Institute's Cancer Ontology, an OWL version of the NCI Thesaurus, is made up of more than 500,000 RDF triples, 17,000 concepts, and is roughly 35 megabytes. Or consider the RDF/XML serialization of UniProt (Universal Protein Resource) KB which is about 134,000,000 triples and 8 gigabytes.

[9] A fragment identifier, according to [10],

> allows indirect identification of a secondary resource by reference to a primary resource and additional identifying information. The secondary resource may be some portion or subset of the primary resource, some view on representations of the primary resource, or some other resource defined or described by those representations.

[10] See [10], section 3.3.1.

is made visible in the browser window. Fragment identifiers do not allow user-agents to manipulate subresources directly; they are, rather, processed by the user-agent only after the resource representation has been retrieved. In fact, HTTP does not even transmit the fragment identifier as part of the `Request-URI`. Origin servers never see the fragment identifier component of the URI.

HTTP does, however, provide an extensible mechanism for user-agents to interact with a range of the resource representation, using the `Range` header. The only range-unit that is explicitly described by the HTTP/1.1 RFC is "bytes". User-agents may address one or more byte ranges in the retrieved resource representation. The bytes range-unit is useful for some IMTs; for example, retrieving a byte-range of a URI that returns a binary image as its representation. The HTTP specification breaks transparency and encourages caches to combine byte ranges under certain validity conditions. The bytes range-unit is obviously much less useful for Semantic Web resources because it isn't appropriate for directly addressing and manipulating *logical* or *semantic* subresources.

3 XPointer and HTTP

The XPointer Framework [9] provides an extensible mechanism for addressing XML subresources. The XPointer Working Group defined a core set of XPointer addressing schemes — [4], [8], and [5] — in addition to the fundamental XPointer Framework. Since XPointer is an extensible scheme, we can define new XPointer schemes and adopt existing XPointer schemes for addressing subresources of specific XML vocabularies. For example, SVG [6] defines a scheme for interpreting fragment identifiers as logical views into a rendered SVG document.

Of the existing W3C-standardized XPointer schemes, the `element()` scheme is too weak and the `xpointer()` scheme is so complex that it is rarely implemented. But it is simple enough to declare and implement new schemes. Such schemes can be crafted specifically for the specific nature of particular resource types. Consider an XPointer addressing scheme, `xpath()`, that would facilitate addressing subresources using the widely adopted XPath Recommendation [3]. Or consider a scheme, `isbn()`, that would facilitate addressing metadata resources about books by ISBN identifier.

We want to shape our XPointer schemes such that they will serve dual use: not only will they provide a data access mechanism for subresources, but they will also make it possible for authorized user-agents to directly manipulate those subresources. A card catalog resource, for example, could be maintained by interchanging XML fragments that represent metadata about books in a library's collection.

3.1 Server-side XPointer

How can we use the XPointer Framework and HTTP to address and manipulate resources and subresources *directly*? The key pieces include an XPointer scheme and three HTTP headers: `Accept-Range`, `Range`, and `Content-Type`.

First, the origin server uses the `Accept-Range: xpointer` header-key/value pair to signal its support for the XPointer Framework. Second, the client sends a subresource request via the `Range: [xpointer-scheme] = [pointer-parts]` header-key/value pair. Third, the server provides a partial content response using the `Content-Type: multipart/mixed` header-key/value pair.

Our main proposal is very simple: the user agent creates an HTTP `Range` request header using the fragment identifier, which makes the fragment identifier a visible part of the request sent to the origin server. That way XPointer schemes may be processed by the origin server, and so user agents do not have to retrieve an entire representation just in order to apply an XPointer scheme, the intention of which may very well be to work with only a subresource. In sum, the only new requirement imposed by our scheme on a user agent is that, when it prepares an HTTP request, it must create a `Range` header, the value of which is the fragment identifier.[11] `xpointer` in the `Range` header specifies the range-unit. The range-unit signals the server as to the appropriate interpretation of the range-specifier, which in this case is `element`.

Consider the following example. The user agent's request:

```
GET /resource HTTP/1.1
Host: example.org
Accept: text/xml
Range: xpointer=element(a12)
```

The origin server's response:

```
HTTP/1.1 206 Partial Content
Content-Type: multipart/mixed; boundary="simple boundary"

--simple boundary

Content-type: text/xml; charset=ISO-8859-1

<bar>Hello</bar>

--simple boundary--
```

When the server uses an XPointer processor to evaluate the XPointer expression — the `element(a12)` range-specification from the `Range` header — against the negotiated representation[12] of the current state of the resource, the result is a node set consisting of the single XML element whose ID type attribute has

[11] User agents must also be able to process the response by recognizing the HTTP status code 206 Partial Content and must be able to handle the `multipart/mixed` IMTs.

[12] Recall that the semantics of any particular fragment identifier are defined in terms of a particular IMT and are applied to the negotiated, retrieved representation of the resource. When a user agent is smart about the use of fragment identifiers, it can correctly choose the right fragment identifier for the desired content type, which

the value "a12". The response, then, contains a single XML fragment, which is the serialization of that XML element. However, since there could have been multiple matched XML subresources in the general case, the individual XML subresources are returned as the body parts of a multipart/mime response entity. The HTTP status code is a 2xx series, indicating success; but the 206 status code is used to indicate that partial content is being returned by the server in response to the `Range` header included by the client.

3.2 Addressing Multiple Subresources

Consider an XPointer scheme named `xpath()`. Since it's not an official XPointer scheme, it *must* be placed into an explicit namespace using the XPointer `xmlns()` scheme. This provision for explicit namespaces on XPointer schemes is partially responsible for XPointer's extensibility.

The user agent's request:

```
GET /mydoc HTTP/1.1
Host: example.org
Accept: text/xml
Range: xpointer=xmlns(xp=http://example.org/xptr/xpath)xpath(//bar)
```

The origin server's response:

```
HTTP/1.1 206 Partial Content
Content-Type: multipart/mixed; boundary="simple boundary"

--simple boundary

Content-type: text/xml; charset=ISO-8859-1

<bar id="a12">Hello</bar>

--simple boundary

Content-type: text/xml; charset=ISO-8859-1

<bar id="a13">World</bar>

--simple boundary--
```

is done using the HTTP Accept request header. However, if the user agent attempts to apply a fragment identifier that has no semantics for the retrieved IMT-typed representation, then the evaluation of the fragment identifier will fail. So, given a link to an SVG resource using the `svgView()` fragment identifier, a user agent that does not support SVG may negotiate for a JPEG representation of the resource. The `svgView()` scheme is not defined against the JPEG IMT, and thus the user agent, for example, a browser, will not be able to establish the correct viewport onto the retrieved image.

The response entity now contains one body part for each matched node in the addressed XML resource. Further, the MIME body parts are presented to the client in document order, just as they would be identified if you applied that XPath expression directly to the negotiated `text/xml` representation of that resource.

4 RDF Query Language and XPointer

There is a large and growing number of query languages for RDF. The W3C has chartered the Data Access Working Group to create a recommendation for both an RDF query language and a data access protocol. Since XPointer is an extensible framework, we can choose any RDF query language — including, but not limited to one standardized by the Data Access Working Group — and develop a notation for that query language that is consistent with the XPointer Framework. Once that is done, we can encode a query in a URI and use the techniques described above to address subresources — for example, subgraphs of an RDF graph, all the classes in an OWL ontology, the creator of an RDF snippet — identified by that query language.

Consider an XPointer scheme, `rdql()`, for RDQL [11]. Since `rdql()` isn't an officially specified W3C XPointer scheme, we use the `xmlns()` scheme to setup the namespace context so that `rdql()` is correctly recognized by the XPointer processor. Once the namespace context has been established, we set the resulting XPointer expression as the value of the `Range` header. When the request is received by the server it will use an XPointer processor to evaluate that expression on behalf of the client and ship back the results.

The user agent's request:

```
GET /myStore HTTP/1.1
Accept: application/rdf+xml
Range: xmlns(q=http://www.myorg.org/xpointer-scheme/rdql)
       rdql(
         SELECT (?x foaf:mbox ?mbox)
          WHERE (?x foaf:name "John Smith")
                (?x foaf:mbox ?mbox)
          USING foaf FOR <http://xmlns.com/foaf/0.1/>
       )
```

The origin server's response:

```
HTTP/1.1 206 Partial Content
Content-Type: application/rdf+xml

<rdf:RDF xmlns:rdf="...">...</rdf:RDF>
```

Notice that in this case multiple subresources are being returned using an atomic, as opposed to a multipart, IMT. We can do this for three reasons: First,

we are using a logical as opposed to syntactic addressing scheme. Second, we are assuming that any subresources of an RDF graph are also RDF graphs, which is to say that you can't address anything more simple than an RDF triple. Third, any RDF graph can be interchanged as a single `application/rdf+xml` document.

5 RDF, OWL, and XPointer

Our proposal enables user agents to delegate the evaluation of the fragment identifier to the origin server. The client is essentially declaring that it only needs those subresource(s) that are actually addressed by the fragment identifier. `rdf()` and `owl()` XPointer schemes would allow for appropriate kinds of server (and, subsequently, client-side) processing of Semantic Web resources. But what about the reflective modeling questions raised above?

5.1 Reflective RDF Modeling and Management

Consider a Semantic Web application which allows RDF graphs to be collected and aggregated. One approach to managing such an application is to include provenance information about each bit of RDF in the snippet itself. That is, for each RDF graph, it would contain explicit triples asserting, for example, the creator of the snippet, as well as the date the snippet was submitted and the date it was updated. Thus, a query for the metadata about a particular RDF snippet might retrieve every assertion with the snippet's URI as the subject; but that approach has some obvious problems.

A more tractable solution is to store the metadata about each snippet separately from the RDF graph about which that metadata makes assertions. Storing the metadata about an RDF graph outside the graph has some obvious advantages; suppose, for example, that we want to assert that the content of some snippet is untrustworthy or outdated. Presumably that characterization of the assertions in the graph is not intended to apply to the characterization itself. That is, we might very well want to say things about the whole graph which were not meant to be said of the graph's metadata.

This point exposes a modeling problem for RDF data and the Semantic Web generally. We want to treat RDF documents as web resources, one of the general uses of which is to make assertions about other resources. But we also want to be able to make assertions about RDF graphs, *qua* Web resources, themselves. We want, therefore, some measure of reflective or second-order expressiveness about all web resources, including the ones that are RDF and OWL. One approach to achieving this kind of reflective expressivity is to establish a URI convention for the metadata of a graph — by, say, appending a "/rdf" URI path segment to any URI. Setting aside the widespread breakage that might impose, it's not a very elegant solution. It is far more reasonable to create an XPointer scheme, say `metadata()` that, in conjunction with our server-side XPointer proposal, would

be taken by Semantic Web origin servers to be a query or retrieval of information *about* a resource rather than *of* the resource.

This approach is similar to the URIQA [13] proposal in that both approaches avoid the needless proliferation of URIs. We, however, want to have a URIRef for the metadata about a resource as well. The URIQA approach — to create new HTTP methods, most notably `MGET` — effectively hides the switch from resource to metadata about the resource in the request.[13] Which says nothing, of course, about the non-technical intractability of standardizing a new set of core HTTP methods.

5.2 OWL Partial Imports

OWL ontologies are usually encoded, published, and exchanged as RDF/XML documents. One way in which the needs of the Semantic Web, *quo* a KR-like system, are not fully served by the typical implementation of HTTP/1.1 is the importation of third party OWL ontologies. We think the proposed Semantic Web resource protocol described herein can address some of the problems associated with OWL partial imports.

In order to understand the utility of our proposal for OWL imports, it's important to review its semantics. Using `owl:imports` in ontology A to import ontology B has the effect of merging the set of RDF triples which constitute the two ontologies. One of the problems with multi-ontology Semantic Web applications is selective use of a term or terms from another ontology. For example, consider an OWL ontology about biological research, which may need to use a few terms from a very large ontology about proteins, but which, for practical reasons, does not want to simply `owl:imports` the very large ontology. Since the only semantics of `owl:imports` is something like "graph concatenation", the choices seem to be either import the entire large ontology, do not import any of it, or shadow some of its terms by declaring equivalent classes in the host ontology. Each of these options has real disadvantages.

But using the techniques described herein, we can specify *partial imports* of resources which are, in fact, subresources addressed and retrieved by means of RDF or OWL or even arbitrary XQuery or XPath queries. By providing a URI that contains an XPointer scheme — for example, the `rdql` scheme described above — as the value of an `owl:imports`, we can specify a complex query to filter a published OWL ontology to a subresource that only contains the triples we need.

Such filtered, partial imports may well be semantically problematic in light of the original ontology creator's intention; but whatever semantic perturbations

[13] URIQA not only proposes three new HTTP methods, `MGET`, `MPUT`, and `MDELETE`, but it also requires clients and origins servers to support a new HTTP header, `URIQA-uri`: "it is recommended that when submitting a URIQA request where the request URI contains a fragment identifier, the full URI should be redundantly specified using the special HTTP message header URIQA-uri." The proposal herein has the relative advantage of using only extant headers and URI serialization forms.

are created by a partial import are no different in principle than the perturbations that may be created by an ordinary import. The publisher of an OWL ontology has no control over what happens to it once it is imported into an arbitrary ontology.

A partial import has the advantage of neither touching the original creator's URI space nor requiring some kind of shadow resource or document. It does require coordination of XPointer schemes between the user agent and origin server, however; though, as we point out in the next section, this does not impose a very high burden of implementation. Furthermore, we can imagine future XPointer schemes that would allow us to address subresources of OWL ontologies that are OWL specific; for example, addressing and retrieving the subresource of an ontology that is a particular class and all its subclasses.

6 Implementation Experience

As a test of our claims herein we have developed software to support two XPointer schemes for server-side processing.

6.1 Partial OWL Imports in Pellet

The Pellet OWL Reasoner [12] is a Java DL reasoner implemented, using the tableaux algorithm, by members of the UMD Mindlab, including one of the authors of this paper. The developers of Pellet support a subset of RDQL queries as a kind of conjunctive ABox query. As a proof of the concepts described in this paper, we deployed Pellet in a simple HTTP application environment wherein it used the `Accept-Range: xpointer` HTTP header to advertise its ability to process XPointer URIs.

We were then able to address and retrieve representations of subresources of the very large NCI cancer ontology using RDQL and Pellet. We were able to do this with very simple HTTP user agents; for example, we used the Unix command line tool `wget`, which allows the insertion of an arbitrary HTTP header into every request. With this environment we are able to create complex RDQL queries, use those queries to address and retrieve subresources of the NCI cancer ontology, and we're able to do this with very little implementation burden. The user agent implementation is very minimal, and the origin server burden is equally minimal.

6.2 RSS Querying

RSS 1.0 [1] is an RDF vocabulary for representing and exchanging typed web site event streams. These event streams, known colloquially as "channels", are published on the Web and referred to by URI. Because it is an RDF vocabulary, the kinds of events that it can serialize are limited only by RDF itself. As further proof of the concepts described herein, we implemented an XPointer scheme, `rss`, that allowed querying of an RSS channel in two ways.

First, the `rss` scheme supports queries by list slice semantics; that is, taking the channel to be a list of events, you can address a subresource of the channel by indexing into the list in the same way as you index a list in a high-level programming language. Because we used Python as our implementation language, we simply exposed a subset of the Python list slice semantics to the `rss` scheme. For example, `rss(-1)` addresses the last entry of the channel; and `rss(0:5)` addresses the first five channel entries.

Second, we implemented support for arbitrary date range queries. One of the standard properties of the entries of an RSS channel is a date property. For example, `rss(2004-05-05:2004-05-07)` addresses the subresource constituted by all the RSS events occurring on or between 2004-0505 and 2004-05-07.

7 Conclusion and Future Work

The nature of most Semantic Web resources, represented by RDF and OWL documents, is significantly different from World Wide Web resources, represented by HTML and XML documents. The latter are intended for human consumption and syntactic interoperability. The former are KR formalisms and are amenable to logical query languages and KR-sensitive addressing schemes. But the Semantic Web is still supposed to be the Web, which suggests that extensions of existing addressing and protocol technologies are appropriate.

We have described herein one such extension, which has as its goal the creation of a Semantic Web resource protocol using the extensible XPointer addressing scheme and some features of HTTP/1.1. The benefits of this proposal include *scalability*, derived from XPointer processing at the origin server rather than by the client, as well as *extensibility*, which allows for addressing and interaction models that are sensitive to RDF and OWL. In particular our proposal offers a way of handling the resource/resource-metadata distinction without the needless propagation of URIs, as well as offering some traction for the problems of OWL imports.

We conclude that the proposal described herein is practically realizable on a large scale, such realization being primarily a matter of multiparty agreement as to the semantics of particular XPointer schemes. We anticipate more detailed analyses of the semantics of RDF and OWL-specific XPointer schemes in conjunction with HTTP methods like `DELETE`, `PUT`, `POST`, `LINK` and `UNLINK`.

References

1. G. Beged-Dov, D. Brickley, R. Dornfest, I. Davis, L. Dodds, and et. al. RDF Site Summary (RSS) 1.0. http://web.resource.org/rss/1.0/spec.
2. T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifier (URI): Generic Syntax. IETF Network Working Group Internet Draft, http://gbiv.com/protocols/uri/rev-2002/rfc2396bis.html.
3. J. Clark and S. DeRose. XML Path Language (XPath) Version 1.0. W3C Recommendation 16 November 1999 http://www.w3.org/TR/xpath/.

4. S. DeRose, E. Maler, and R. Daniel Jr. XPointer xpointer() Scheme. W3C Working Draft 19 December 2002 http://www.w3.org/TR/xptr-xpointer/.
5. S. J. DeRose, R. Daniel Jr., E. Maler, and J. Marsh. XPointer xmlns() Scheme. W3C Recommendation 25 March 2003 http://www.w3.org/TR/xpointer-xmlns/.
6. J. Ferraiolo, F. Jun, and D. Jackson. Scalable Vector Graphics (SVG) 1.1 Specification. W3C Recommendation 14 January 2003 http://www.w3.org/TR/SVG11/.
7. N. Freed and N. Borenstein. Multipurpose Internet Mime Extensions (MIME) Part Two: Media Types. IETF Network Working Group RFC 2046, http://www.ietf.org/rfc/rfc2046.txt.
8. P. Grosso, E. Maler, J. Marsh, and N. Walsh. XPointer elemnet() Scheme. W3C Recommendation 25 March 2003 http://www.w3.org/TR/xptr-element/.
9. P. Grosso, E. Maler, J. Marsh, and N. Walsh. XPointer Framework. W3C Recommendation 25 March 2003 http://www.w3.org/TR/xptr-framework/.
10. I. Jacobs. Architecture of the World Wide Web. W3C Working Draft 9 December 2003 http://www.w3.org/TR/webarch/.
11. A. Seaborne. RDQL - A Query Language for RDF. W3C Member Submission 9 January 2004, http://www.w3.org/Submission/RDQL/.
12. E. Sirin and B. Parsia. Pellet OWL Reasoner. http://www.mindswap.org/2003/pellet/.
13. P. Stickler. The Nokia URI Query Agent Model. http://sw.nokia.com/uriqa/URIQA.html.