Proceedings of the
3nd International Workshop on

Scripting for the Semantic Web (SFSW 2007)

Co-located with 4rd European Semantic Web Conference
June 3-7, 2007, Innsbruck, Austria.

Workshop Co-Chairs' Message

SFSW 2007 - Workshop on Scripting for the Semantic Web

Scripting languages such as PHP, JavaScript, Ruby, Python, Perl, JSP and ActionScript are playing a central role in current development towards flexible, lightweight web applications following the AJAX and REST design paradigms. These languages are the tools of a generation of web programmers who use them to quickly create server and client-side web applications. Many deployed Semantic Web applications from the FOAF, RSS/ATOM, blog and wiki communities, as well as many innovative mashups from the Web 2.0 and Open Data movements are using scripting languages and it is likely that the process of RDF-izing existing database-backed websites, wikis, weblogs and CMS will largely rely on scripting languages.

The workshop brings together developers of the RDF base infrastructure for scripting languages with practitioners building applications using these languages. Last years Scripting for the Semantic Web workshop in Budva, Montenegro focused on giving an overview about the support for Semantic Web technologies within scripting languages. The special focus of this year's workshop is the role of scripting languages in the process of populating the Web with linked RDF data as well as to showcase innovative scripting applications that consume RDF data from the Web. The focus especially of the included Scripting Challenge is to show how Web 2.0 applications and mashups can benefit from Semantic Web technologies.

We would like to thank the organizers of ESWC conference for supporting the workshop. We especially thank all members of the SFSW program committee for providing their expertise and giving elaborate feedback to the authors and T-Systems for providing the Scripting Challenge prize. Last but not least, we hope that you will enjoy the workshop and the whole conference.

Chris Bizer, Freie Universität Berlin, Germany
Sören Auer, University of Pennsylvania, USA
Gunnar Aastrand Grimnes, DFKI, Germany
Tom Heath, Open University, UK

SFSW 2007 Program Committee

- David Aumüller, Universität Leipzig, Germany
- Danny Ayers, Independent Author, Italy
- Dave Beckett, Yahoo!, USA
- Uldis Bojars, DERI, Ireland
- Dan Brickley, Semantic Web Vapourware, UK
- Richard Cyganiak, Freie Universität Berlin, Germany
- Stefan Decker, DERI, Ireland
- Stefan Dietze, KMi, The Open University, UK
- Leigh Dodds, Ingenta, UK
- Edd Dumbill, Useful Information Company, UK
- Frank Fuchs-Kittowski, Frauenhofer Gesellschaft - ISST, Germany
- Daniel Krech, University of Maryland, USA
- Peter Mika, Vrije Universiteit Amsterdam, The Netherlands
- Libby Miller, @Semantics, UK
- Claudia Müller, University of Potsdam, Germany
- Benjamin Nowack, appmosphere web applications, Germany
- Alberto Reggiori, @Semantics, Italy
- Sebastian Schaffert, salzburg research, Austria
- Vlad Tanasescu, KMi, The Open University, UK
- Elias Torres, IBM, USA
- Denny Vrandecic, AIFB, Universität Karlsruhe, Germany
- Gregory Williams, University of Maryland, USA

Table of Contents

The Web Mashup Scripting Language Profile

Marwan Sabbouh, Jeff Higginson, Caleb Wan, Salim Semy, Danny Gagne

The MITRE Corporation
202 Burlington Rd.
Bedford, Massachusetts 01730
ms@mitre.org

Abstract. This paper provides an overview of the Web Mashup Scripting Language (WMSL) and discusses the WMSL-Profile. It specifies the HTML encoding that is used to import Web Service Description Language (WSDL) files and metadata, in the form of mapping relations, into a WMSL web page. Furthermore, the WMSL-Profile describes the conventions used to parse the WMSL pages. It is envisioned that these WMSL pages scripted out by end-users using an easy-to-use editor will allow mashups to be created quickly to integrate Web services. The processing of these WMSL pages will be accomplished automatically and transparently to generate aligned ontologies sufficient for interoperability.

Key Words: Semantics, Scripting, Ontologies, Mapping Relations, Web Services, Mashup

1 Introduction

The Web Mashup Scripting Language (WMSL) [1] enables an end-user ("you") working with a browser—not even needing any other infrastructure, to quickly write mashups that integrate any Web services on the Web. The end-user accomplishes this by writing a web page that combines HTML, metadata in the form of mapping relations, and small piece of code, or script. The mapping relations enable not only the discovery and retrieval of other WMSL web pages, but also affect a new programming paradigm that abstracts many programming complexities from the script writer. Furthermore, the WMSL web pages written by disparate end-users ("you") can be harvested by crawlers to automatically generate the concepts needed to build aligned ontologies sufficient for interoperability [4].

Despite many advances in Semantic Web technologies, such as OWL [6] and RDF [7], we have not observed widespread adoption of these technologies that was once anticipated. In comparison, a number of lightweight technologies such as Microformats [8], Ajax [9], RSS [10], and REST [11] enjoy substantial momentum and support from Web communities. The apparent reason for the success of these technologies is that they are effective in addressing needs and fairly simple to use.

We believe the adoption of Semantic Web technologies has been slow largely because they involve heavyweight infrastructure and substantial complexities. Adding to these issues are the multiple competing standards in Semantic Web Services [12], e.g. OWL-S [13] and WMSO [14], and how they can be harmonized with existing W3C standards. Therefore, one key issue to address is this: Can we adopt Semantic Web technologies in a manner sufficient to our needs that is lightweight and much less complex for Web communities to use? Our previous research has clearly indicated that light semantics are sufficient in many applications, and can be used as a transitional step to rich semantics. For example, we concluded in [3] that we can achieve workflow automation from the pair-wise mappings of data models and from their mapping to some shared context, regardless of whether OWL\RDF, XML Schemas, or UML is used to describe the data models.

Another challenge to widespread adoption of rich semantics is the lack of social processes for the design of ontologies as is in the case for Folksonomies or in the social tagging case. Despite these difficulties, we cannot escape the fact that semantics are absolutely needed to enable automated reasoning on the web, or to enable information exchange in the enterprise. How can we promote Web user participation in using semantics without requiring deep understanding of ontology?

We believe WMSL, when combined with existing schemas such as WSDL files, offers sufficient semantics for many applications. That is, WMSL leverages all the semantics that exist in XML schemas while offering the facilities to assert further semantics that may be missing from XML Schemas. This positions WMSL as the glue that takes in XML schemas and yields formal ontologies. Since WMSL is HTML and scripting, it therefore has Web scale. Furthermore, WMSL is lightweight, and can be run from a browser. Also, our solution enables a light SOA approach where anyone can write a WMSL script to implement a mashup in support of information sharing requirements. Finally, WMSL can automatically generate the semantics needed to index and search structured data as is done with free text today.

In the next section, we provide an overview of WMSL. In section 3, we use an example to describe the encoding conventions of WMSL followed by its parsing conventions in section 4. In section 5, we relate this approach to the literature, discuss its implications, and point out the next steps to conclude the paper.

2 WMSL Overview

The WMSL script is divided into four blocks to contain different types of statements:

1. Imports of Web Service Description Language (WSDL) files [2], schemas, ontologies, and other WMSL scripts
2. Alignments of entities and concepts
3. Workflow statements
4. Mediation statements

Each of these blocks can be encoded either in HTML or a script. For the purpose of this paper, we discuss the WMSL-Profile: the encoding of the import and alignment blocks in the HTML of a WMSL web page. That is, we describe the conventions of encoding and of parsing the WMSL-Profile. We also describe the automatic generation of aligned ontologies from the WMSL-Profile. It is envisioned that WMSL pages are created by end-users with the help of an easy-to-use editor, and the parsing of the WMSL pages, which yields the aligned ontologies, is accomplished automatically and transparently.

Figure 1 shows a WMSL page for a use case presented in [3] and [4]. The use case discusses the integration of two air flight systems: Air Mobility (AM), and Air Operations (AO). The AM system is responsible for many different types of missions including: mid-air refueling, the movement of vehicles, and the tasking of Air Force One. The AO system is primarily concerned with offensive and defensive missions. Each system was developed independently and built for the specific needs of the users. In both systems, a mission is represented as a set of position reports for a particular aircraft.

```
<html>
        <head profile="http://mitre.org/wmsl/profile">
                <title>WSML Use Case</title>
                <base href=" http://mitre.org/owl/1.1/"/>
                <link rel="schema.AM" type="text/xml"
                        href="http://www.mitre.org/xsd/1.1/AM#"/>
                <link rel="schema.AO" type="text/xml"
                        href="http://www.mitre.org/xsd/1.1/AO#"/>
        </head>
        <body>
                <dl class="owl-equivalentClass">
                        <dt><a href="AM#CallSign">AM#CallSign</a></dt>
                        <dd><a href="AO#CallSignName">AO#CallSignName</a></dd>
                </dl>
                <dl class="owl-sameAs">
                        <dt><a href="AM#A10A">AM#A10A</a></dt>
                        <dd><a href="AO#A010A">AO#A010A</a></dd>
                </dl>
                <dl class="mappings-match">
                        <dt><a AM#AircraftType">AM#AircraftType</a></dt>
                        <dd><a AO#AircraftType">AO#AircraftType</a></dd>
                </dl>
                <dl class="mappings-hasContext">
                        <dt><a href="AO#AOCoord">AO#AOCoord</a></dt>
                        <dd><a href="position#Coord-GEODETIC-WGE">
                                position#Coord-GEODETIC-WGE</a></dd>
                </dl>
                <dl class="mappings-hasRelation">
                        <dt><a href="position#Coord-UTM-WGE"></a></dt>
                        <dd><a href="position#UTM"></a></dd>
                </dl>
                <dl class="rdfs-subclassOf">
                        <dt><a href="AM#A10A"></a></dt>
                        <dd><a href="AM#AircraftType"></a></dd>
                </dl>
        </body>
</html>
```

Fig. 1. A Sample WMSL Profile for the AM-AO Use Case

In this scenario these two systems will be integrated so that the AO system can be kept apprised of all the AM missions. To accomplish this integration a WMSL page will be created. As stated earlier we will be focusing only on the import and alignment blocks. First, the WMSL imports the WSDL files of the AM and AO, and the WSDL of shared context which is the GeoTrans translator service that translates between geo-coordinate systems. These imports yield the aligned ontologies necessary to reconcile syntactic, structural, and representational mismatches between the AM and the AO schemas as was demonstrated in [3] and [4]. Moreover, these imports also yield the ontological description of web services necessary for their automatic invocation and for handling their response—that aspect will not be addressed here but in a future paper.

Then, the WMSL uses six mapping relations to align entities between the AM and AO schemas and for their mappings to the WSDL of Geotrans [5]. Our previous work [3] and [4] provides us a basis and insight to identify a minimal set of mapping relations for reconciling mismatches between data models in most cases if not all. The mapping patterns which are discussed in greater detail in our previous work are shown in Figure 2. The minimal set includes only these mapping relations:

owl:equivalentClass *owl:sameAs* *rdfs:subclassOf*
hasMatch *hasContext* *hasRelation*

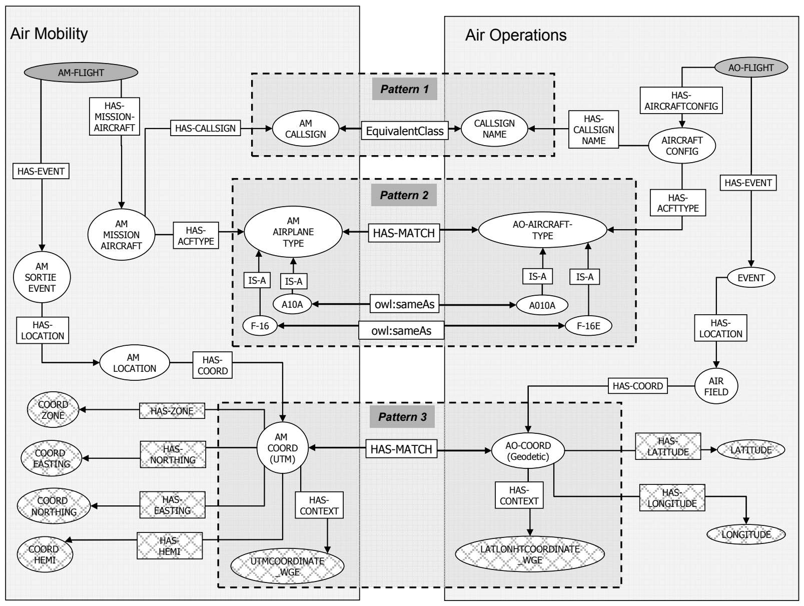


Fig. 2. Aligned Ontologies for the AM-AO Use Case

The first three relations are used in accordance with the specifications that they were taken from. The *hasMatch*, and *hasContext* relations are needed in order to resolve structural, syntactic, and representational mismatches between the legacy schemas. The *hasRelation* establishes a generic relationship between a subject and an object. To conclude this section we highlight the fact that the imports of the WSDL files and the existence of these mapping relations in the WMSL enable an open-source/collaborative model of building aligned ontologies sufficient for interoperability.

3 WMSL-Profile Specifications

3.1 Encoding of the Schema Declarations Using the Header Block

We now specify the conventions used in encoding the WMSL-Profile. Other encodings are certainly possible, and we would welcome help for defining better encoding scheme. We have chosen HTML in which to define WMSL because of its already widespread acceptance and familiarity to Web communities. And there is no need to introduce new syntax and tags when the familiar standard HTML tags would suffice for WMSL.

To direct the user agent to follow the WMSL conventions in parsing this document, we use the standard HTML profile attribute of the head tag as shown in Figure 3.

```
<head profile=http://mitre.org/wmsl/profile>
```

Fig. 3. Use of the Profile Attribute

To declare the WSDL files employed by the integration, we use a method compliant with that used by the embedded RDF specification as well as the method used by the Dublin Core to embed metadata in HTML using the *link* and *meta* tags. Specifically, we use the *rel*, *type* and *href* attributes of the *link* tag. The general pattern is shown in Figure 4 followed by examples in Figures 5 and 6.

```
<link rel="schema.prefix" type=″MIME Content Type″ href="uri" />
<link rel="schema.AM" type="text/xml“ href="http://www.mitre.org/xsd/1.1/AM#"/>
<link rel="schema.AO" type="text/xml" href="http://www.mitre.org/xsd/1.1/AO#"/>
```

Fig. 4. Import of the WSDL Files Using the Link Tag

The above statements also declare a schema prefix that can be used later in the HTML. Next we declare the schema of the mapping relations. Notice that we adopt the type value “application/rdf+xml” to indicate an ontology file.

```
<link rel="schema.map" type="application/rdf+xml“
       href="http://www.mitre.org/mappings/1.1/mappings#"/>
```

Fig. 5. Use of the Type Attribute

Next, we specify the handles for using relations from the RDFS and OWL specifications. In the next section we will show how these handles are used.

```
<link rel="schema.owl" type="text/html" href="http://www.w3.org/2002/07/owl#"/>
<link rel="schema.rdfs" type="text/html"
       href="http://www.w3.org/2000/01/rdf-schema#"/>
```

Fig. 6. Use of Schema Handles

3.2 Encoding of the Mapping Relations in the Body Tag

The technique used for alignment requires six mapping relations used in three design patterns. As stated earlier the mapping relations are: *owl:equivalentClass*, *owl:sameAs*, *rdfs:subclassOf, hasMatch*, *hasContext*, *hasRelation*. In this section we define the encoding of the mapping patterns in HTML. For these relations the *class* attribute of *DL* tag is used in combination with the *anchor*, *DT* and the *DD* tags. The interpretation of the encoding is also addressed in the next section.

The encoding of the *owl::equivalentClass* relation between two entities is shown in Figure 7. Note the use of the OWL prefix in the class attribute of the *DL* tag, this is the same prefix that was declared in the *rel* attribute of the *link* tag of Figure 6.

```
<dl class="owl-equivalentClass">
   <dt><a href="http://mitre.org/owl/1.1/AM#CallSign">AM#CallSign</a></dt>
   <dd><a href="http://mitre.org/owl/1.1/AO#CallSignName">AO#CallSignName</a></dd>
</dl>
```

Fig. 7. Encoding of the owl:equivalentClass

The encoding of the *owl:sameAs* relation between two entities is similar to that of the *owl:equivalentClass*, and is shown in Figure 8.

```
<dl class="owl-sameAs">
    <dt><a href="http://mitre.org/owl/1.1/AM#A10A">AM#A10A</a></dt>
    <dd><a href="http://mitre.org/owl/1.1/AO#A010A">AO#A010A</a></dd>
 </dl>
```

Fig. 8. Encoding of the owl:sameAs

Next, we demonstrate the encoding of the *hasMatch* and *hasContext* relations. The first example shown in Figure 9 specifies that the triple AM AircraftType *hasMatch* the AO AircraftType. The second example shown in Figure 9 specifies the triples AOCoord *hasContext* Coord-GEODETIC-WGE, and AMCoord *hasContext* Coord-UTM-WGE.

```
<dl class="mappings-hasMatch">
    <dt><a href="http://mitre.org/owl/1.1/AM#AircraftType">AM#AircraftType</a></dt>
    <dd><a href="http://mitre.org/owl/1.1/AO#AircraftType">AO#AircraftType</a></dd>
</dl>
<dl class="mappings-hasContext">
    <dt><a href="http://mitre.org/owl/1.1/AO#AOCoord">AO#AOCoord</a></dt>
    <dd><a href="http://mitre.org/owl/1.1/position#Coord-GEODETIC-WGE">
              position#Coord-GEODETIC-WGE</a></dd>
    <dt><a href="http://mitre.org/owl/1.1/AM#AMCoord">AM#AMCoord</a></dt>
    <dd><a href="http://mitre.org/owl/1.1/position#Coord-UTM-WGE">
              position#Coord-UTM-WGE</a></dd>
</dl>
```

Fig. 9. Encoding of the hasMatch and hasContext

The encoding of the *rdfs:subclassOf* relation to specify that the aircraft A10A is a subclass of AircraftType is shown in Figure 10.

```
<dl class="rdfs-subclassOf">
    <dt><a href="http://mitre.org/owl/1.1/AM#A10A"></a></dt>
    <dd><a href="http://mitre.org/owl/1.1/AM#AircraftType"></a></dd>
</dl>
```

Fig. 10. Encoding of the rdfs:subclassOf

Finally, the *hasRelation* mapping relation, shown in Figure 11, is encoded in HTML to specify that that the entity Coord-UTM-WGE has two generic relations with UTM, and WGE. A generic relation is a genetic property where the name is not significant.

```
<dl class="mappings-hasRelation">
    <dt><a href="http://mitre.org/owl/1.1/position#Coord-UTM-WGE"></a></dt>
    <dd><a href="http://mitre.org/owl/1.1/position#UTM"></a></dd>
    <dt><a href="http://mitre.org/owl/1.1/position#Coord-UTM-WGE"></a></dt>
    <dd><a href="http://mitre.org/owl/1.1/position#WGE"></a></dd>
</dl>
```

Fig. 11. Encoding of the hasRelation

4 Automatic Generation of the Aligned Ontologies When Parsing the WMSL-Profile

In our previous work, we have demonstrated that given the aligned ontologies of Figure 2, we can automatically translate an instance of the Air Mobility (AM) to an instance of the Air Operations (AO). How can we obtain the aligned ontologies of Figure 2? The WSDL files specified in the import block of the WMSL-Profile can be converted into ontologies. However, the WSDL files may not contain all the entities necessary to enable the information exchange; hence, we use the mappings in the WMSL-Profile to specify, or create, the missing semantics. The end result is that the parsing of the WMSL-Profile yields the aligned ontologies sufficient for integration. (For now, we will ignore the case where the schema declared in the WMSL may themselves be ontologies.) More details on generating aligned ontologies are described in the next sections.

4.1 Generating Ontologies from WSDL Files

We start building the ontology by leveraging the semantics already existing in the WSDL file. To do that, we create mapping patterns between XML schema primitives and the OWL/RDF vocabulary; the complete set of patterns will be discussed in a future paper. For example, class membership is derived from the XML schema sequence (*xs:sequence*), and restrictions on properties from the minOccurs/maxOccurs attributes of the *xs:sequence* tag. Figure 12 shows a snippet of xml schema and Figure 13 shows the corresponding ontology of it. Since XML schema does not contain property names, we use a generic relationship in the RDF triple. From our previous work we found that the property name of the triple within an ontology does not play a role in the reasoning necessary to reconcile syntactic, structural, and representational mismatches between data models.

```
<xs:complexType name="AircraftConfigType">
       <xs:sequence>
              <xs:element name="AircraftType" type="xs:string"
                                          minOccurs="0" maxOccurs="1"/>
              <xs:element name="CallSignName" type="xs:string"
                                          minOccurs="0" maxOccurs="1"/>
       </xs:sequence>
```

Fig. 12. Snippet of XML Schema

```
<owl:Class rdf:ID="AIRCRAFTCONFIG">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#HAS-AIRCRAFTTYPE"/>
      <owl:minCardinality rdf:datatype="&xsd;string">0</owl:minCardinality>
      <owl:maxCardinality rdf:datatype="&xsd;string">1</owl:maxCardinality>
      <owl:allValuesFrom rdf:resource="#AIRCRAFTTYPE"/>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#HAS-CALLSIGNNAME"/>
      <owl:minCardinality rdf:datatype="&xsd;string">0</owl:minCardinality>
      <owl:maxCardinality rdf:datatype="&xsd;string">1</owl:maxCardinality>
      <owl:allValuesFrom rdf:resource="#CALLSIGNNAME"/>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

Fig. 13. OWL Statements Corresponding to the XML Schemas of Figure 12

After the ontologies have been generated from the WSDL files, we proceed to augment these ontologies with semantics from the mapping relations in the WMSL-Profile.

4.2 Augmenting the Ontologies with Semantics Derived from Mapping Relations

This step is illustrated by generating the semantics from the mappings presented in the previous sections. The *owl:equivalentClass* relation that was presented in Figure 7 yields the following OWL (Figure 14).

```
<owl:Class rdf:ID="CALLSIGNNAME">
       <owl:equivalentClass rdf:resource="&AM;CALLSIGN"/>
</owl:Class >
```

Fig. 14. OWL Statements Corresponding to the WMSL of Figure 7

The *owl:sameAs* and *rdfs:subclassOf* mapping relations, presented in Figure 8 and Figure 10 respectively, yield the following OWL in the AM ontology and the AO ontology (Figure 15).

```
<owl:Class rdf:ID="A10A">
       <rdfs:subClassOf>
              <owl:Class rdf:about="&AM;AIRCRAFTTYPE"/>
       </rdfs:subClassOf>
       <owl:sameAs rdf:resource="&AO;A010A" />
</owl:Class>

<owl:Class rdf:ID="A010A"/>
```

Fig. 15. OWL Statements Corresponding to the WMSL of Figure 8

The *hasRelation* mapping relation shown in Figure 11, yields the following OWL (Figure 16).

```
<owl:Class rdf:ID="Coord-UTM-WGE">
       <rdfs:subClassOf>
              <owl:Restriction>
                     <owl:onProperty rdf:resource="#HAS_RELATION"/>
                     <owl:someValuesFrom rdf:resource="&position;UTM"/>
              </owl:Restriction>
       </rdfs:subClassOf>
       <rdfs:subClassOf>
              <owl:Restriction>
                     <owl:onProperty rdf:resource="#HAS_RELATION"/>
                     <owl:someValuesFrom rdf:resource="&position;WGE"/>
              </owl:Restriction>
       </rdfs:subClassOf>
</owl:Class>
```

Fig. 16. OWL Statements Corresponding to the WMSL of Figure 10

Note that the OWL above is inserted into a position ontology that was included in the WMSL-Profile. In similar fashion, the remaining mapping relations yield OWL definitions. When we finish the parsing of the WMSL-Profile, the aligned ontologies are created as shown in the Figure 2 above.

5 Relation to the Literature and Future Work

The ideas presented in this paper draw on the proliferation of semantic matching techniques found in Semantic Web Services [13] literature. We abstract one such technique and formulate it in HTML. In the process, we demonstrated how WMSL is used to leverage existing schemas to produce ontologies. This allows us to think of WMSL as the glue between schemas and ontologies. WMSL can potentially enable matching between schemas irrespective of their formalisms. Today, techniques to embed semantics in HTML are emerging, but with a different purpose from that of WMSL. For example, the hCard Microformat is used to embed contact information in HTML pages. RDFa [15] serves to embed metadata such as those defined by the Dublin Core, in HTML. In contrast to RDFa, WMSL is designed to embed mapping relations in HTML. Another key distinction between the approach presented here and the Microformats is that WMSL builds on schemas, and not text pages. Moreover, the embedding of the mapping relations in HTML serves to promote crosswalks for the purpose of building ontologies. This is a key differentiator from the tagging phenomenon that is so relevant in Folksonomies or the annotation technique enabled by SAWSDL. That is, crosswalks may prove to be as significant to the structured data sources as tags are to resources. Furthermore, since anyone can publish WMSL for existing WSDLs, we conclude that WMSL enables an open source model for building ontologies.

In conclusion, this paper describes how metadata in the form of mapping relations are embedded in HTML. We also described the parsing conventions of WMSL by a user agent. Our next steps, which will be described in a separate paper, are to demonstrate how the mapping relations abstract workflow composition and to make available libraries for enabling the execution of WMSL web pages.

References

[1] Sabbouh, M., Higginson, J., Semy, S., Gagne, D. Web Mashup Scripting Language. Available at : http://semanticweb.mitre.org/wmsl/wmsl.pdf
[2] Webservice Description Language (WSDL) 1.1, http://www.w3.org/TR/2001/NOTE-wsdl-20010315, June 2006
[3] Gagne D., Sabbouh M., Powers S., Bennett S. Using Data Semantics to Enable Automatic Composition of Web Services. IEEE International Conference on Services Computing (SCC 06), Chicago USA. (Please see the extended version at:http://tinyurl.com/28svgr)
[4] Sabbouh M. et al. Using Semantic Web Technologies to Enable Interoperability of Disparate Information Systems, MTR: http://www.mitre.org/work/tech_papers/tech_papers_05/05_1025/
[5] Schroeder, B., & Sabbouh, M. (2005). Geotrans WSDL, http://www.openchannelfoundation.org/orders/index.php?group_id=348, The Open Channel Foundation
[6] Web Ontology Language (OWL), World Wide Web Consortium, http://www.w3.org/2004/OWL
[7] Resource Description Framework (RDF), World Wide Web Consortium, http://www.w3.org/rdf/
[8] More information on Microformats available at: http://microformats.org/
[9] More information on Asynchronous Javascript and XML (AJAX) available at: http://www.ajaxmatters.com/
[10] Really Simple Syndication (RSS) 2.0 specification, available at: http://www.rssboard.org/rss-specification
[11] Fielding, R.T., Architectural Styles and the Design of Network-based Software Architectures, PhD Dissertation in Information and Computer Science, 2000, available at: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm
[12] McIlraith S., &Son T.,Zeng H. (2001). Semantic Web services. In IEEE Intelligent Systems (Special Issue on the Semantic Web)
[13] OWL-S: Semantic Markup for Web Services, http://www.w3.org/Submission/OWL-S/, November, 2004
[14] Web Service Modeling Ontology (WSMO), http://www.w3.org/Submission/WSMO/, June 2005
[15] RDFa Primer 1.0, available at: http://www.w3.org/TR/xhtml-rdfa-primer/

A Performance and Scalability Metric for Virtual RDF Graphs

Michael Hausenblas[1], Wolfgang Slany[2], and Danny Ayers[3]

[1] Institute of Information Systems and Information Management,
JOANNEUM RESEARCH, Steyrergasse 17, 8010 Graz, Austria
`michael.hausenblas@joanneum.at`
[2] Institute for Software Technology,
Graz University of Technology, Inffeldgasse 16b, 8010 Graz, Austria
`wsi@ist.tugraz.at`
[3] Independent Author, Italy
`danny.ayers@gmail.com`

Abstract. From a theoretical point of view, the Semantic Web is understood in terms of a stack with RDF being one of its layers. A Semantic Web application operates on the common data model expressed in RDF. Reality is a bit different, though. As legacy data has to be processed in order to realise the Semantic Web, a number of questions arise when one is after processing RDF graphs on the Semantic Web. This work addresses performance and scalability issues (PSI), viz. proposing a metric for *virtual RDF graphs on the Semantic Web*—in contrast to a local RDF repository, or distributed, but native RDF stores.

1 Motivation

The Semantic Web is—slowly—starting to take-off; as it seems, this is mainly due to a certain pressure stemming from the Web 2.0 success stories. From a theoretical point of view the Semantic Web is understood in terms of a stack. The Resource Description Framework (RDF) [1] is one of the layers in this stack, representing the common data model of the Semantic Web.

However, practice teaches that this is not the case, in general. In the perception of the Semantic Web there exists a tremendous amount of legacy data. This is, HTML pages with or without microformats[4] in it, relational databases (RDBMS), various XML applications as Scalable Vector Graphics (SVG)[5], and the like. These formats are now being absorbed into the Semantic Web by approaches as Gleaning Resource Descriptions from Dialects of Languages (GRDDL) [2], RDFa in HTML [3], etc.

Take Fig. 1 as an example for a real-world setup of a Semantic Web application. There, a Semantic Web agent operates on an RDF graph with triples that actually originate from a number of sources, non-RDF or 'native' RDF alike.

[4] `http://microformats.org/`
[5] `http://www.w3.org/Graphics/SVG/`

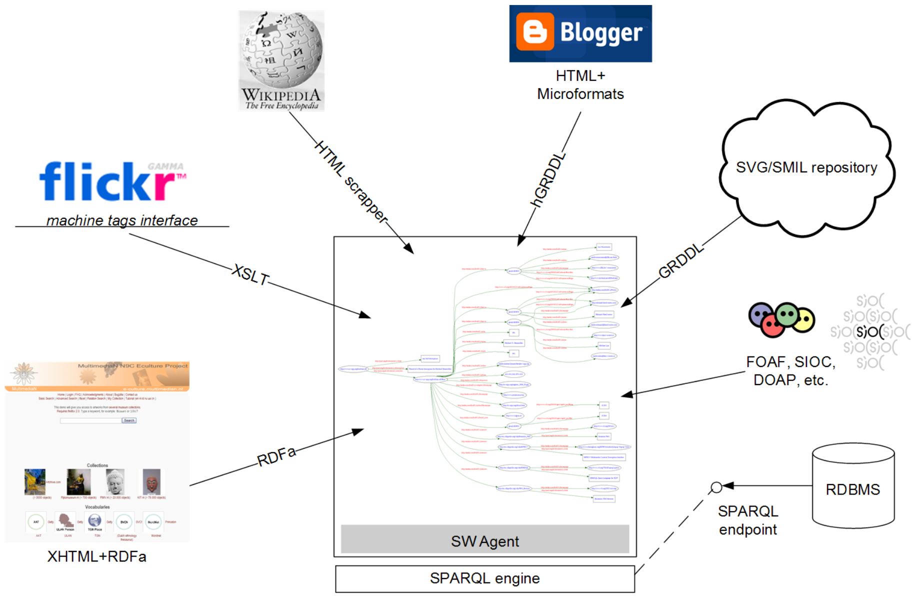


Fig. 1. A real-world setup for a Semantic Web application.

The Semantic Web agent operates on its local RDF graph in terms of performing for example a SPARQL Protocol And RDF Query Language [4] query to accomplish a certain task. While from the point of view of the SPARQL engine it might not be of interest where triples come from and how they found their way into the local RDF graph, the Semantic Web agent—and finally the human user who has instructed the agent to carry out a task—may well be interested in how long a certain operation takes.

But how do the triples arrive in the local RDF graph? Further, how do the following issues influence the performance and the scalability of the operations on the local RDF graph:

– The number of sources that are in use;
– The types of sources, as RDF/XML, RDF in HTML, RDBMS, etc.;
– Characteristics of the sources: a fixed number of triples vs. dynamic, as potentially in case of a SPARQL end point.

This paper attempts to answers these questions. We first give a short overview of related and existing work, then we discuss and define virtual RDF graphs, their types, and characteristics. How to RDF-ize the flickr Web API[6], based on the recently introduced *machine tags*[7] feature, serves as a showcase for the proposed metric. Finally we conclude on the current work and sketch directions for further investigations.

[6] `http://www.flickr.com/services/api/`
[7] `http://www.flickr.com/groups/api/discuss/72157594497877875/`

2 Related and Existing Work

The term scalability is used differently in diverse domains—a generic definition is not available. For selected domains, various views on scalability are available [5, 6]. Bondi [7] recently elaborated on scalability issues. He considers four types of scalability: *load scalability*, *space scalability*, *space-time scalability*, and *structural scalability*.

Practice-oriented research regarding RDF stores has been reported in the SIMILE[8] project [8], and in a joint W3C-EU project, Semantic Web Advanced Development for Europe [9]. In the Advanced Knowledge Technologies (AKT) project, 3store [10]—a scalable RDF store based on Redland[9]—was used. A framework for testing graph search algorithms where there is a need for storage that can execute fast graph search algorithms on big RDF data is described in [11].

At W3C, RDF scalability and performance is an issue[10], though the scope is often limited to local RDF stores. There are also academic events that address the scalability issue, such as the International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS)[11] which took place the second time in 2006.

Some research already exists regarding distributed RDF stores. Though the focus is on distributed RDF repositories, it is always assumed that one is dealing with 'native' RDF sources. An excellent example is the work of Stuckenschmidt et. al. [12]. They present an architecture for optimizing querying in distributed RDF repositories by extending an existing RDF store (Sesame). Cai and Frank [13] report on a scalable distributed RDF repository based on a peer-to-peer network—they propose RDFPeers that stores each triple at three places in a multi-attribute addressable network by applying globally known hash functions. From the Gnowsis project, a work has been reported that comes closest to ours: Sauermann and Schwarz [14] propose an adapter framework that allows for integrating data sources as PDFs, RDBMS, and even Microsoft Outlook.

To the best of our knowledge no research exists that addresses the issue of this paper, i.e. performance and scalability issues of virtual RDF graphs on the Semantic Web. Regarding the distributed aspect, the authors of [12] listed two strong arguments, namely i) the *freshness*, viz. in not using a local copy of a remote source frees one from the need of managing changes, and ii) the gained *flexibility*—keeping different sources separate from each other provides a greater flexibility concerning the addition and removal of sources. We subscribe to this view and add that there are a number of real-world use cases which can only be addressed properly when taking distributed sources into account. These use cases are to be found in the news domain, stock exchange information, etc.

[8] `http://simile.mit.edu`
[9] `http://librdf.org/`
[10] `http://esw.w3.org/topic/TripleStoreScalability`, and `http://esw.w3.org/topic/LargeTripleStores`
[11] `http://www.cs.vu.nl/~holger/ssws2006/`

3 Virtual RDF Graphs

To establish a common understanding of the terms used in this paper, we first give some basic definitions. We describe what a Semantic Web application in our understanding is, and have a closer look at virtual RDF graphs. The nature of virtual RDF graphs, and their intrinsic properties are discussed in detail.

Definition 1 (Semantic Web application). *A **Semantic Web application** is a software program that meets the following minimal requirements:*

1. *It is based on, i.e. utilises HTTP*[12] *and URIs*[13]*;*
2. *For human agents, the primary presentation format is (X)HTML*[14]*, for software agents, the primary interface formats are based on Web services*[15] *and/or based on the REST approach [15];*
3. *The application operates on the Internet; the number of concurrent users is undetermined.*
4. *The content used is machine readable and interpretable; the data model of the application is RDF [1].*
5. *A set of formal vocabularies—potentially based on OWL [16]—is used to capture the domain of discourse. At least one of the utilised vocabularies has to be proven **not** to be under full control of the Semantic Web application developer.*
6. ***Non-mandatory**, SPARQL [4] is in use for querying, and RIF [17] for representing, respectively exchanging rules.*

The restriction that a Semantic Web application is expected to operate on the Internet is to ensure that Intranet applications that utilise Web technologies are **not** understood as Semantic Web applications in the narrower sense. This is a matter of who controls the data and the schemes rather than a question of the sheer size of the application.

Definition 1 requires that a Semantic Web application operates on the RDF data model, which leads us to the virtual RDF graph, defined as follows.

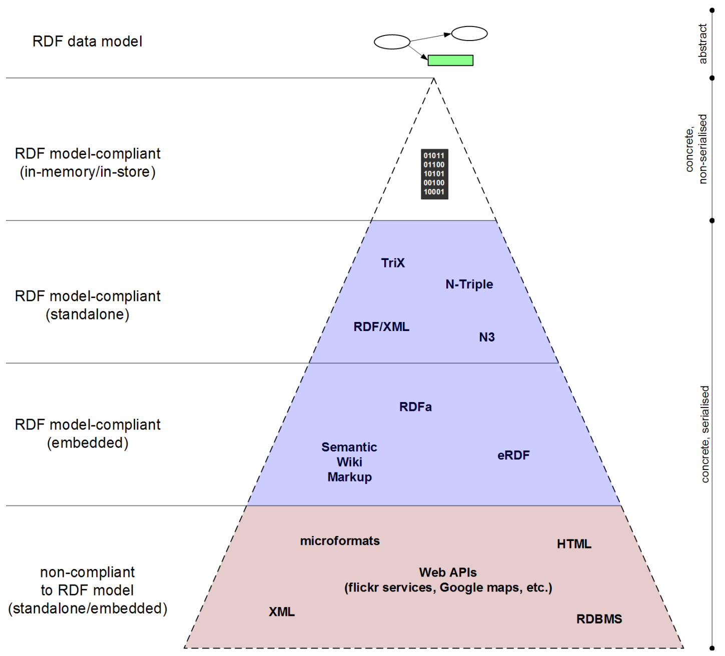
Definition 2 (Virtual RDF Graph). *A **virtual RDF graph** (vRDF graph) is an RDF graph local to a Semantic Web application that contains triples from potentially differing, non-local sources. The primary function of the vRDF graph is that of enabling CRUD*[16] *operations on top of it. The following is trivially true for a vRDF graph:*

1. *it comprises actual source RDF graphs (henceforth sources), with N_{src} being the number of sources;*
2. *each source S^i_{src} contributes a number of triples T^i_{src} to a vRDF graph, with $0 < i \leq N_{src}$;*
3. *The vRDF graph contains $\sum T^i_{src}$ triples.*

[12] `http://www.w3.org/Protocols/rfc2616/rfc2616.html`
[13] `http://www.ietf.org/rfc/rfc2396.txt`
[14] `http://www.w3.org/html/wg/`
[15] `http://www.w3.org/2002/ws/`
[16] **c**reate, **r**ead, **u**pdate and **d**elete—the four basic functions of persistent storage

3.1 Types Of Sources

Triples may stem from sources that utilise various representations. In Fig. 2 the representational properties of the sources are depicted, ranging from the RDF model[17] to non model-compliant sources.



RDF data model
RDF model-compliant (in-memory/in-store)
01011 01100 10101 00100 10001
RDF model-compliant (standalone)
TriX
N-Triple
RDF/XML
N3
RDF model-compliant (embedded)
RDFa
Semantic Wiki Markup
eRDF
non-compliant to RDF model (standalone/embedded)
microformats
HTML
Web APIs (flickr services, Google maps, etc.)
XML
RDBMS
abstract
concrete, non-serialised
concrete, serialised

Fig. 2. The RDF representation pyramid.

The two middle layers of the pyramid denote representations that are RDF model-compliant and have a serialisation, hence may be called *native RDF*. Representations that do have a serialisation, but are not RDF model-compliant, may be referred to as *non-RDF sources*. We therefore differentiate:

Standalone, RDF model-compliant Representations. These type of sources, for example RDF/XML, can be stored, transmitted, and processed on their own. For example an in-memory Document Object Model (DOM) representation of an RDF/XML document can be built by utilising a SAX parser.

Embedded, RDF model-compliant Representations. Sources of this type, as RDFa in HTML [3] or eRDF[18], need a host to exist; their representation is only defined in the context of this host. Here, the triples are produced by applying a transformation.

[17] `http://www.w3.org/TR/rdf-concepts/#section-data-model`
[18] `http://research.talis.com/2005/erdf/wiki/Main/RdfInHtml`

Representations non-compliant to the RDF model. The majority of the data sources on the Web, standalone or embedded, is of this type:

- GRDDL [2] is utilised to 'uncover' RDF in, e.g., HTML. The same applies to microformats that can be RDF-ized using hGRDDL[19];
- An RDBMS that provides for a SPARQL end point [4] can be used to contribute triples[20];
- Syndicated feeds (RSS 2.0, Atom[21]) are another source;
- From a HTML page without explicit metadata, triples may be gathered through screen scrapers (as [18]).

In order to be processed, the serialisation is required to be "converted" from a representation with a concrete syntax into an in-memory representation. This conversion may occur through applying a transformation[22], or by parsing the specified syntax.

3.2 Characteristics Of Sources

Besides the type of the source, a further distinction w.r.t. the number of triples from a source can be made. We distinguish between fixed sized sources and undetermined—or dynamic—sized source.

Take for example a Wiki site that serves as a source for a vRDF graph. Let us assume an HTML scraper is used to generate triples from selected Wiki pages, for example based on a category. The number of resulting triples then is in many cases stable and can be assessed in advanced. In contrast to this, imagine an RDBMS that provides for a SPARQL end point—the D2R Server[23] is a prominent example for this—as an example for a dynamic source. Based on the query, the number of triples varies. A border case are social media sites, as blogs. They are less dynamic than data provided by a SPARQL end point but constantly changing and growing as more comments come in.

4 A Metric for virtual RDF Graphs

In this section we describe a performance and scalability metric that helps a Semantic Web application developer to assess her/his vRDF graph. A showcase for a non-native RDF source is then used to illustrate the application of the metric.

The execution time of an operation on a vRDF graph is influenced by a number of factors, including the number of sources in a vRDF graph N_{src}, the overall number of triples $\sum T^i_{src}$, and the type of the operation. The metric proposed in Definition 3 can be used to assess the performance and scalability of an vRDF graph.

[19] `http://www.w3.org/2006/07/SWD/wiki/hGRDDL_Example`

[20] Though, this source may also be considered as being native in terms of the interface.

[21] `http://bblfish.net/work/atom-owl/2006-06-06/AtomOwl.html`

[22] see `http://esw.w3.org/topic/ConverterToRdf`, and also `http://esw.w3.org/topic/CustomRdfDialects`

[23] `http://sites.wiwiss.fu-berlin.de/suhl/bizer/d2r-server/`

Definition 3 (Execution Metric). *The overall execution time for performing a CRUD function (as inserting a triple, performing a SPARQL ASK query, etc.) is denoted as t_P; the time for converting a non-RDF source representation into an RDF graph is referred to as t_{2RDF}. The total time delays due to the network (Internet) transfer are summed up as t_D; the time for the actual operation performed locally is denoted as t_O. Obviously,*

$$t_P = t_O + t_{2RDF} + t_D$$

The "conversion time vs. the overall execution time"-ratio is defined as

$$coR = \frac{t_{2RDF}}{t_P}$$

To illustrate the above introduced metric, a showcase has been set up, which is described in the following.

A Showcase For Non-Native RDF Sources: PSIMeter[24]. The showcase demonstrates the application of the metric by RDF-izing the flickr API. Three different methods have been implemented; the non-native RDF Source used in the PSIMeter showcase is the information present in the machine tags. The goal for each of the three methods is to allow a Semantic Web agent to perform a SPARQL construct statement, as for example:

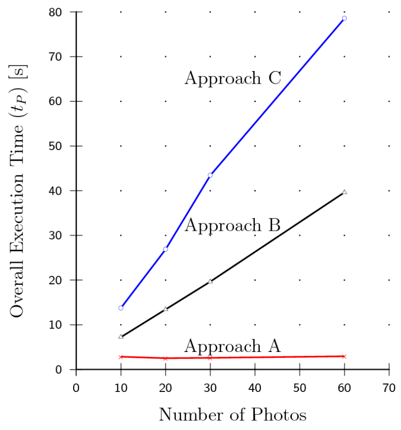
```
CONSTRUCT { ?photoURL dc:subject ?subject }
WHERE     { ?photoURL dc:subject ?subject.
             FILTER regex(?subject, "XXX", "i") }
```

The experiments to compare the three approaches were run on a testbed that comprised up to 100 photos from a single user, along with annotations in the form of machine tags, up to 60 in total. Machine tags were selected as the source due to their straightforward mapping to the RDF model. The three methods for constructing the vRDF graph work as follows:

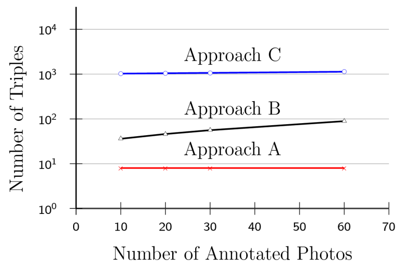
1. *Approach A* uses the search functionality of the flickr API[25] in a first step to retrieve the IDs of photos tagged with certain machine tags. In a second step the flickr API is used to retrieve the available metadata for each photo. Finally the result of the two previous steps is converted into an RDF representation, locally;
2. *Approach B* uses the flickr API to retrieve all public photos[26] firstly. It then uses a local XSL transformation to generate the RDF graph;
3. *Approach C* retrieves all public photos, as in Approach B. Then, for each photo an external service[27] is invoked to generate the RDF graph.

[24] available at `http://sw.joanneum.at:8080/psimeter/`
[25] `http://www.flickr.com/services/api/flickr.photos.search.htm`
[26] `http://www.flickr.com/services/api/flickr.people.getPublicPhotos.html`
[27] `http://www.kanzaki.com/works/2005/imgdsc/flickr2rdf`

Firstly, for a *fixed query*, `dc:subject=marian`, the overall execution time t_P has been measured depending on the number of photos (Fig. 3(a)). In Fig. 3(b), the size of the vRDF graph in relation to the number of annotations, with a fixed number of photos, is depicted.



(a) Overall execution time dependency on number of photos.



(b) vRDF graph size dependency on number of annotated photos.

Fig. 3. PSIMeter: Metric for a fixed query.

The second experiment focused on the impact of the *query type* on the overall execution time.

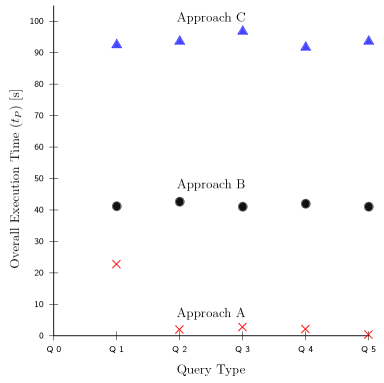


Fig. 4. PSIMeter: Metric in dependency on Query Type.

The results of the second experiment are depicted in Fig. 4, with the according queries listed in Table 1.

Reference	Query
Q1	`dc:subject=dummy`
Q2	`dc:title=NM2`
Q3	`dc:title=`
Q4	`geo:location=athens`
Q5	`dc:dummy=test` (empty match)

Table 1. Query Types

Note, that more than 80% of the photos were tagged with `dc:subject=dummy`, hence Q1 exhibits an exception.

Another finding of the experiment, was that in all evaluation runs, *coR* tended towards 1 (ranging from 0.95 to 0.99), viz. most of the time the system was busy converting the data to RDF, and only a small fraction was dedicated to the actual operation of applying the SPARQL construct statement.

5 Conclusion

When building Semantic Web applications, it is not only important to operate on distributed RDF graphs by means of virtue, but also to question how the triples in a vRDF graph where produced. We have looked at variables that influence the performance and scalability of a Semantic Web application, and proposed a metric for vRDF graphs. As all types of sources must be converted into an in-memory representation in order to be processed, the selection of the type of sources is crucial. The experiments highlight the importance to use existing search infrastructure, as the flickr search API in our case, as far as possible, hence converting only results to RDF.

Another generic hint is to avoid conversion cascades. As long as there exists a direct way to create an in-memory representation, this issue does not play a vital role. Though, regarding performance issues, this is of importance in case an intermediate is used to create the in-memory representation, as with, e.g., the hGRDDL approach.

Finally, the incorporation of dynamic sized sources is a challenge one has to carefully implement. This is a potential area for further research in this field.

6 Acknowledgements

The research reported in this paper was carried out in two projects: the "Knowledge Space of semantic inference for automatic annotation and retrieval of multimedia content" (K-Space) project[28], partially funded under the 6th Framework Programme of the European Commission, and the "Understanding Advertising" (UAd) project[29], funded by the Austrian FIT-IT Programme.

[28] `http://kspace.qmul.net/`
[29] `http://www.sembase.at/index.php/UAd`

References

1. G. Klyne, J. J. Carroll, and B. McBride. RDF/XML Syntax Specification (Revised). `http://www.w3.org/TR/rdf-concepts/`, 2004.
2. D. Connolly. Gleaning Resource Descriptions from Dialects of Languages (GRDDL). `http://www.w3.org/TR/2007/WD-grddl-20070302/`, 2007.
3. M. Hausenblas and B. Adida. RDFa in HTML Overview. `http://www.w3.org/2006/07/SWD/RDFa/`, 2007.
4. E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. `http://www.w3.org/TR/rdf-sparql-query/`, 2006.
5. M. D. Hill. What is scalability? *SIGARCH Comput. Archit. News*, 18(4):18–21, 1990.
6. P. Jogalekar and M. Woodside. Evaluating the Scalability of Distributed Systems. *IEEE Trans. Parallel Distrib. Syst.*, 11(6):589–603, 2000.
7. A. B. Bondi. Characteristics of scalability and their impact on performance. In *WOSP '00: Proceedings of the 2nd International Workshop on Software and Performance*, pages 195–203, New York, NY, USA, 2000. ACM Press.
8. R. Lee. Scalability Report on Triple Store Applications. `http://simile.mit.edu/reports/stores/`, 2004.
9. D. Beckett. Deliverable 10.1: Scalability and Storage: Survey of Free Software/Open Source RDF storage systems. Technical report, Semantic Web Advanced Development for Europe (SWAD-Europe), IST-2001-34732, 2002.
10. S. Harris and N. Gibbins. 3store: Efficient bulk RDF storage. In *Proc. 1st International Workshop on Practical and Scalable Semantic Systems (PSSS'03), Sanibel Island*, pages 1–15, 2003.
11. M. Janik and K. Kochut. BRAHMS: A WorkBench RDF Store and High Performance Memory System for Semantic Association Discovery. In *Proc. 4th International Semantic Web Conference, ISWC 2005, Galway, Ireland*, pages 431–445, 2005.
12. H. Stuckenschmidt, R. Vdovjak, G.-J. Houben, and J. Broekstra. Index Structures and Algorithms for Querying Distributed RDF Repositories. In *WWW '04: Proc. of the 13th International Conference on World Wide Web*, pages 631–639, New York, NY, USA, 2004. ACM Press.
13. M. Cai and M. Frank. RDFPeers: a scalable distributed RDF repository based on a structured peer-to-peer network. In *WWW '04: Proc. of the 13th International Conference on World Wide Web*, pages 650–657, New York, NY, USA, 2004. ACM Press.
14. L. Sauermann and S. Schwarz. Gnowsis Adapter Framework: Treating Structured Data Sources as Virtual RDF Graphs. In *Proc. 4th International Semantic Web Conference, ISWC 2005, Galway, Ireland*, 2005.
15. R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures.* PhD thesis, University of California, Irvine, 2000.
16. OWL. Web Ontology Language Reference. W3C Recommendation, 10 February 2004.
17. Rule Interchange Format (RIF). `http://www.w3.org/2005/rules/`, 2007.
18. R. Baumgartner, G. Gottlob, M. Herzog, and W. Slany. Annotating the Legacy Web with Lixto. In *Proc. 3rd International Semantic Web Conference, ISWC2004, Hiroshima, Japan*, 2004.

Explorative Debugging for Rapid Rule Base Development

Valentin Zacharias and Andreas Abecker

FZI Research Center for Information Technologies, Haid-und-Neu Strasse 10-14, Karlsruhe 76131, Germany
`zacharias|abecker@fzi.de`

Abstract. We present Explorative Debugging as a novel debugging paradigm for rule based languages. Explorative Debugging allows truly declarative debugging of rules and is well suited to support rapid, try-and-error development of rules. We also present the Inference Explorer, an open source explorative debugger for horn rules on top of RDF.

1 Debugging Semantic Web Rules

Semantic Web rule languages can be an important tool for the rapid development of Semantic Web applications [7]. The large scale use of these languages is, however, currently still hampered by missing tool support for their creation - in particular missing debugging support.
Debugging support for Semantic Web rule bases must address two challenges:

– It is known that web developers have a particularly high percentage of end user programmers [9, 6]; hence development tools for the Semantic Web have to take special care to adjust to end user programmers. For debugging tools this means in particular to support the try and error, rapid, incremental development process often observed with end user programmers [10, 11].
– Rules are declarative programs that describe what is true but not how something is calculated. A debugger must take this into account.

This paper starts with a definition of rules and an introduction into the rule syntax used throughout this paper. It then discusses existing debugging approaches and their limitations. After these have been established, it proposes Explorative Debugging as a better debugging paradigm. It describes the building blocks of Explorative Debugging and shows how they are implemented in the Inference Explorer application.

2 Terminology

To date there is no agreed upon definition or specification of rules for the Semantic Web, also the ideas presented in this paper can be applied to a number of different rule languages. However, in the interest of having a consistent terminology throughout this paper, we define a simple rule language based on the rules used by the well known Jena framework [3] that also forms the basis of the implementation. In this paper only rules that work on data in the form of RDF [1] triples are considered.

A rule base is a finite set of rules. The rules are of the form:

```
Rule_Example:
   (?A,b,c) <- (?A,x,y) (?A,v,w)
```

The part to the left of the ← symbol is the *goal* or *head* of the rule, the part to the right the *body* or *subgoals*. The head of the rule consists of one *atom*, the body of a conjunction of atoms. An atom is a *triple pattern*, a triple of *terms*[1]. A triple pattern is a RDF triple without anonymous nodes but with named variables instead of some terms. A triple pattern is *matched* against an RDF triple by replacing the variables with terms from the triple; the matching succeeds when the variables can be replaced in a way that makes triple pattern and triple identical. A rule is said to *fire* when all atoms in the body of a rule match a triple, with all occurrences of one named variable replaced by the same term.

Rule bases are declarative specifications - they describe *what* is true / can be calculated, not *how* it should be done. A rule base can be evaluated in a number of different ways (e.g. top-down evaluation, bottom-up evaluation, magic sets algorithm) but the result does not depend on the kind of evaluation and rules can be created and understood without knowing the actual evaluation strategy[2].

3 Current Debugging Support For Rules

Existing debugging support for rule based systems can be divided into two groups: procedural debugging and computer controlled debugging.

All deployed debugging tools for rule based programs known to the authors are based on the procedural or imperative debugging paradigm. This debugging paradigm is well known from the world of procedural and object oriented programming and characterized by the concepts of breakpoints and stepping. A procedural debugger offers a way to indicate a rule/rule part where the program execution is to stop and has to wait for commands from the users. The user can then look at the current state of the program and give commands to execute a number of the successive instructions. Of course a rule base does not define an order of execution - hence the order of debugging is based on the evaluation strategy of the inference engine. The steps then, are things like: the inference engine tries to prove a goal A or it tries to find more results for another goal B. This kind of debuggers for rule based systems are available as purely textual

[1] Jena also allows extended triple patterns containing functors and built-ins as atoms, however they are not currently supported in the Inference Explorer; we postpone their discussion until the future work section

[2] Note that this is only partly true for Prolog that mixes procedural and declarative aspects

tracers [12], with a simple graphical user interface [12] or integrated into graphical knowledge acquisition tools [8]. The most sophisticated of these systems was probably the Transparent Prolog Machine [4].

Procedural debuggers break the declarative programming paradigm: they force the user to learn about the working of the inference engine and encourage her to think about the program in an imperative way; also procedural debugging is very difficult to use for anything but simple top-down inference engines. This kind of debugger was necessary for languages like Prolog that mix procedural and declarative aspects; the rule languages proposed for the Semantic web, however, are purely declarative and hence can be debugged in a declarative way.

In computer controlled debugging[3] an algorithm directs the debugging process, the user only needs to answer questions about the expected outcome. The amount of research in this direction is too large to give an overview here; however, in general it can be said:

– These debuggers do not empower the programmer to learn more about the program. Many approaches just propose a change without giving the programmer a chance to test and correct her expectations.
– These debuggers can utilize only information encoded in the program, not the domain knowledge and intuition of the programmer.
– To our best knowledge none of these debuggers has ever proven to work good enough to be used outside of research.

The conclusion then is that current debugging support is unsuited for the rapid development of rules by end user programmers: It either works on the level of the inference engine, becoming hard to use and breaking the declarative programming paradigm. Or it takes control away from the programmer, disempowering the user and becoming imprecise.

4 Explorative Debugging

We propose Explorative Debugging as a better debugging paradigm for rule-based systems. Explorative Debugging works on the declarative semantics of the program and lets the user navigate and explore the inference process. It enables the user to use all her knowledge to quickly find the problem and to learn about the program at the same time. An Explorative Debugger is not only a tool to identify the cause of an identified problem but also a tool to try out a program and learn about its working.

Explorative Debugging puts the focus on rules. It enables the user to check which inferences a rule enables, how it interacts with other parts of the rule base and what role the different rule parts play. Unlike in procedural debuggers the

[3] The authors use computer controlled debugging as a broad term to refer to the areas of Algorithmic Debugging, Declarative Debugging, Automatic Debugging, Why-Not Explanation, Knowledge Refinement, Automatic Theory Revision and Abductive Reasoning

debugging process is not determined by the procedural nature of the inference engine but by the user who can use the logical/semantic relations between the rules to navigate.

An explorative debugger is a rule browser that:

– Shows the inferences a rule enables.
– Visualizes the logical/semantic connections between rules and how rules work together to arrive at a result. It supports the navigation along these connections.
– It allows to further explore the inference a rule enables by digging down into the rule parts.
– Is integrated into the development environment and can be started quickly to try out a rule as it is formulated.

5 Building Blocks Of Explorative Debugging

The main building blocks of Explorative Debugging for rules are: rules, rule firings, rule parts, depends-on-connections and prooftrees. These concepts describe declarative properties of the rule base and take the place of the breakpoints and different stepping commands of procedural debugging. The following sections describe each of these concepts and its role in more detail.

5.1 Rules

The main element for an explorative debugger for rule bases is rules. The debugger is always focused on one rule and can be opened for any rule. Navigation elements allow navigating between rules. Centering a debugger for rule bases around rules sounds self evident, but is not realized in most existing debuggers for rule based systems (these debuggers use the goal the inference engine currently tries to prove as main element).

5.2 Rule Firings

Rule firings are the inferences a rule currently enables. Consider the following rule:

```
Rule_Father:
   (?A,rdf:type,:father) <- (?A,rdf:type,:male) (?A,:parent,?B)
```

and data[4]:

```
  :Abraham  rdf:type  :male;
            :parent   :John.
```

[4] We use the Turtle [2] syntax to represent RDF data, for briefness we omit the prefix declarations

```
  :Johanna  rdf:type  :female;
            :parent   :John.

  :George   :parent   :Michele.
```

In this example there is one rule firing: variable A bound to Abraham and B bound to John; with the current data the rule enables to infer that Abraham is of type father. More formally a rule firing is a set of variable bindings that satisfies the rule body.

The rule firings are the most basic form of checking whether a rule performs according to the expectations of the user. In this example the user probably expected the rule to also infer that George is a father, but sees that this is not the case. Other concepts described below will support the user in investigating this further. Rule firings should be available to the user all the time while creating rules to give quick feedback.

The *current data* - the triples used to calculate the variable bindings - can come from a default data store or from a test setup. For many simple cases the user will have just one example of the data she wants to process with the rules and use this for tests. In such cases this is the default data that can be used to automatically calculate the variable bindings without further ado. In more complex cases a programming environment must support a way to create test cases that contain the appropriate test data that can be used. The rule firings are then displayed with respect to this test data.

5.3 Rule Parts (Firings)

The natural way to examine the unexpected behavior of a rule is to look at the rule parts and to calculate the rule firings for a rule only consisting of a subset of the parts. When, like in the example above, the rule isn't firing for an expected value the user can investigate which rule atom/triple pattern is causing the problem. In this example the user may look at the variable bindings that satisfy the triple pattern (?A,rdf:type,:male), seeing that the expected George isn't included in the list of matching triples; in this way she has narrowed the problem down further.

Rule part firings allow the user to select only a part of the rule body and to look at the variable bindings that satisfy the selected triple patterns.

5.4 Depend-On Connections

The concepts described above only consider a rule and its result as an isolated entity, without the connections to other rules. Depends-on connections are links between rules that indicate that two rules seem to be able to work together, built on top of each other. Consider the following rule base:

```
Rule_Father:
   (?A,rdf:type,:father) <- (?A,rdf:type,:male) (?A,:parent,?B)
```

```
Rule_Parent:
   (?X,:parent,?Y) <- (?Y,:chilt,?X)

Rule_Employer:
   (?X,rdf:type,employer) <- (?X, :owns, ?C) (?A,:employed_at,?C)
```

It can be seen that, with the right data, the rule Father could work on the result of the rule Parent: rule Parent could deduce that A is a parent of B from a child relation and the Father rule could then deduce that A is of type father. In such a case we say that rule Father *depends-on* rule Parent. The depends-on connections are calculated on the rules without consideration for the actual data that is available: a depends-on connection exists independently of the test data. The rule Employer in the example rule base has no depends-on connection to any of the other rules - there exists no set of RDF triples such that the rule Employer could directly work on triples inferred by one of the other two rules.

Formally a depends-on connection exists from rule A to rule B when rule A has a body atom that can be unified with the head atom of B; when rule A has a triple pattern in the body that matches the triple pattern in the head of rule B. Depends-on connections can also be calculated between rule parts and rules: a rule part is a subset of body atoms of a rule. A rule part then depends-on a rule and when one of its atoms is unifiable with the head of the rule.

The depends-on connections calculated this way are heuristic approximations to the rule interactions intended by the user. It is not possible to perfectly identify only those rule interactions that happen with real world data: there can be depends-on connections between rules that never interact in real life.

The importance of the depends-on connection lies in their ability to aid the user in tracking down an error across multiple rules, even if the error prevents any actual interaction between the rules from happening. The following example should illustrate this:

```
  :Abraham  rdf:type  :male.
  :John     :child :Abraham.
```

With the rule base above and these two RDF triples the user expects that rule Father fires for Abraham and John. Opening the debugger for rule Father, however, she sees that the rule doesn't fire. Selecting rule parts she can identify that the pattern (?A,:parent,?B) is causing the problem. This, however, is not the root problem and the depends-on connection supports the user in finding the root cause. Even though there is no actual interaction between the rules a system can show that this rule part depends on rule Parent. The user can use this connection to navigate to the other rule and to discover the root cause: the typo *chilt* in the body of rule Parent. The depends-on links may look like a minor navigation tweak in such a small rule base but becomes an important navigation instrument for large rule bases where the user may not even know about all rules that could be relevant for the current triple pattern.

Obviously there are also errors that impact the depends-on connection: had rule Parent had a typo in the head, there would be no depends-on connection. But the absence of a depends-on link is also an important clue for debugging: a user knowing that a part should depend-on a particular rule has already narrowed down the problem to a mismatch between the current rule part and the head of the other rule. A user expecting a depends-on connection to some rule may be supported by the system in finding rules that almost match.

The importance of depend-on connections cannot easily be overstated for the debugging of rule based systems. Each rule in itself is usually good to understand and a lot of research went into good visualizations for single rules. On the other hand rule interactions and the overall structure of a rule program are usually not readily visible and are often not displayed at all. The depends-on connections and the prooftrees described in the next section will often be the only way to view this structure.

5.5 Prooftrees

Prooftrees represent the actual interactions between rule and RDF facts that lead to results / rule firings. Prooftrees are a tool to quickly identify the reason for unexpected variable firings; for too many results. An example should further clarify this:

```
Rule_Father:
   (?A,rdf:type,:father) <- (?A,rdf:type,:male) (?A,:parent,?B)

Rule_Parent:
   (?X,:parent,?Y) <- (?Y,:child,?X)

Rule_ChildFromRaises:
   (?X,:child,?Y) <- (?Y,:raises,?X)

   :JohnsUncle  rdf:type  :male;
                :raises   :John.
```

Checking the rule firings for the rule "Father" the user is surprised to find that JohnsUncle is inferred to be of type father. A prooftree (or derivation tree) is a representation of the inference process that lead to a particular result and that can aid the user in such a case to find the part of the reasoning chain that acted in an unexpected way. A graphical representation of the actual prooftree for the conclusion (JohnsUncle, rdf:type, father) is shown in Figure 1. It shows that the rule Father concluded this triple and that it depended on the fact that JohnsUncle is of type male and the result of rule Parent, this in turn on another rule and on the triple (JohnsUncle, raises, John). The prooftree allows the user to quickly understand the whole inference process and to pinpoint the rule ChildFromRaises as the root cause of the problem.

The prooftree represents the logical/semantic relationships between rules/ facts and is independent of the actual evaluation algorithm of the inference

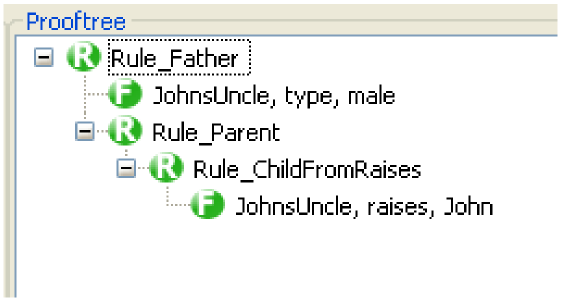
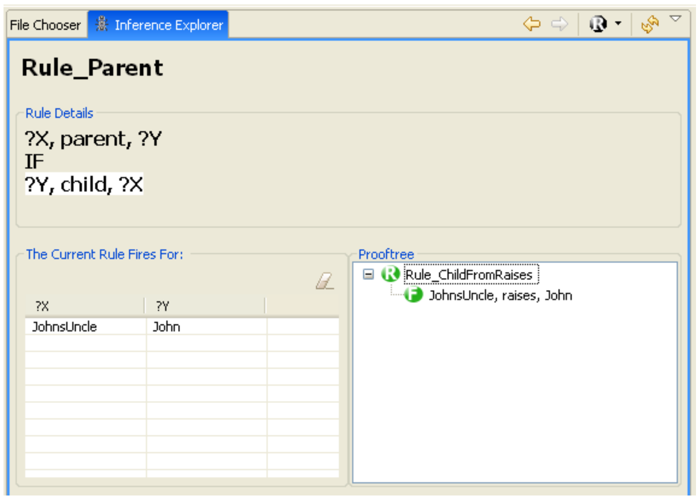


Fig. 1. *Prooftree for the triple (JohnsUncle, rdf:type, father).*

engine. When a rule instantiation in the prooftree is listed as dependent on a fact it says that this fact is necessary for this prooftree to exist. The current result will not be concluded, unless this fact is present[5] The inference engine algorithm can be said to search for and discover the prooftrees, but the prooftrees themselves are defined on the semantics of rules.

6 The Inference Explorer

The Inference Explorer is a debugger for RDF rules that implements the explorative debugging paradigm and that integrates all the building blocks for truly declarative debugging described in the section before.



```
?X, parent, ?Y
IF
?Y, child, ?X
```

?X	?Y
JohnsUncle	John

Fig. 2. *User interface of the Inference Explorer*

[5] Unless, of course, there are multiple prooftrees for one result

The user interface of the Inference Explorer (see Figure 2) consists of three main areas: the rule details in the top, the result view bottom left and the details view in the bottom right. Both the result view and the details view change depending on what the user has selected.

The rule details view shows a textual description of the currently selected rule. The user can select parts of the rules to get more information about these and the rule parts are shown in different colors to highlight unsatisfiable triple patterns.

The result view to the bottom left of the debugger shows the rule firings, the values that satisfy the rule body. When the user selects a rule part in the rule details view the result view changes to display the variable bindings that satisfy this rule part.

The details view to the bottom left changes greatly in response to the currently selected element in the debugger.

- When nothing or a rule part is selected, the details view shows the depends-on connections between the current rule/rule part and other rules in the rule base. The user can click on any rule listed there to open it in the debugger.
- When a result line is selected in the result view, the details views shows the prooftree for this result. This configuration is shown in the screenshot in Figure 2. The user can click on any rule in the prooftree to open the debugger for it.

On the top of the debugger are navigation elements to navigate to arbitrary rules based on their names and elements to move *back* and *forward*, akin to similar buttons in browsers. Other buttons allow to hide URI's before the hash symbol and to reload all data and rules from the (possible changed) files. A second view allows to select the RDF and rules files used.

The InferenceExplorer is currently implemented as a stand-alone Eclipse-RCP [5] program. For RDF storage and rule parsing it uses the Jena Framework. For inferences it uses or own Try![6] inference engine - a simple backward chaining inference engine specifically created to support the debugging of rules. It forfeits speed and memory footprint in order to be easy to understand and to have internal data structures that can be translated to give a human understandable representation of the inference engine's state. Its is created to also support more complex debugging applications beyond the Inference Explorer presented here.

The InferenceExplorer and the Try! inference engine are available as open source and can be downloaded from http://code.google.com/p/trie-rules/.

7 Programming Embedded Debugging - Future Work

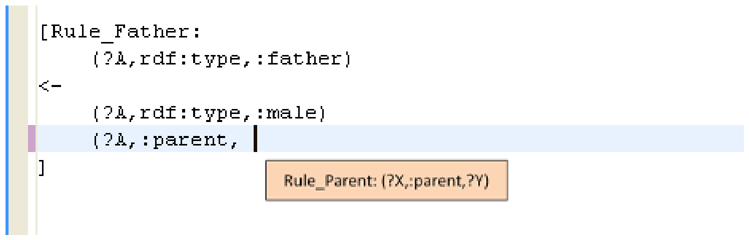
Traditionally, programming and debugging are seen as clearly separate activities: some part of the program is written, a test is created, run and finally the running test may be debugged. The concepts of Explorative Debugging, however, allow

[6] The complete name is TRIE - the Transparent RDF Inference Engine

to naturally integrate some part of the debugging activity into programming and can thereby facilitate faster feedback cycles.

The prerequisite for programming embedded debugging is the definition of *default data*. Default data is some set of RDF triples that is used to evaluate the rules against. This does not exclude the possibility to have other sets of test data, but there needs to be a default set.

With default data it is possible to automatically open the debugger for a rule while it is created by the user. As soon as the user creates a syntactically correct rule the debugger can evaluate this rule in the background and display to the user what this rule infers, given the other rules and the default data. Giving the user an immediate feedback on her rule as it is created.



```
[Rule_Father:
    (?A,rdf:type,:father)
<-
    (?A,rdf:type,:male)
    (?A,:parent,
]
```

Fig. 3. *Mock-up of the integration of depends-on connections into a rule editor.*

Another concept of Explorative Debugging that can be utilized during programming are the depends-on connections. A sophisticated rule editor could immediately display the rules a triple pattern could depend-on as it is created. The mock-up in figure 3 shows how that may look like for a text based rule editor. While the user creates a body atom the one rule she is alerted to a matching head pattern in a different rule. Such integration of the depends-on connections allow the user to quickly get feedback on her assumptions about likely interactions between the rules she is creating.

8 Conclusions and Future Work

The concepts of rules, rule parts, rule firings, depends-on connections and proof-trees allow the debugging of rules purely on the declarative level. The novel debugging paradigm Explorative Debugging brings them together to allow debugging that truly takes into account the declarative nature of rules. Explorative Debugging is well suited to support short feedback loops and to support iterative development.

Our implementation of the Explorative Debugging ideas, The Inference Explorer, is the most advanced debugger for RDF rules available.

For the future we plan to extend the Try! inference engine and to integrate the debugger with editors for RDF and rules. Currently the Try! inference engine is very limited in that it does not support functors, built-ins and anonymous node - we plan to add support for these in the coming weeks. Another problem are rule cycles that can lead the inference engine to run infinitely - we will add algorithms to detect and explain them to the user. After that the further development will be based on feedback from users of the debugger.

References

1. D. Beckett. RDF/XML syntax specification. W3C recommendation, RDF Core Working Group, World Wide Web Consortium, 2004.
2. D. Beckett. Turtle - terse RDF triple language. http://www.dajobe.org/2004/01/turtle/, 2004. (accessed 2007-03-30).
3. J.J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson. Jena: Implementing the semantic web recommendations. Technical report, Hewlett Packard, 2003.
4. M. Eisenstadt, M. Brayshaw, and J. Paine. *The Transparent Prolog Machine: visualizing logic programs*. Intellect Books, 1991.
5. Eclipse Foundation. Rich client platoform. http://wiki.eclipse.org/index.php/Rich_Client_Platform, 2007. (accessed 2007-03-21).
6. W. Harrison. The dangers of end-user programming. *IEEE Software*, July/August:5–7, 2004.
7. M. Kifer, J. de Bruijn, H. Boley, and D. Fensel. A realistic architecture for the semantic web. In *RuleML*, pages 17–29, 2005.
8. ObjectConnections. Common knowledge. http://www.objectconnections.com/, 2007. (accessed 2007-02-13).
9. M. B. Rosson, J. Balling, and H. Nash. Everyday programming: Challenges and opportunities for informal web development. In *Proceedings of the 2004 IEEE Symposium on Visual Languages and Human Centric Computing*, 2004.
10. J. Ruthruff and M. Burnett. Six challenges in supporting end-user debugging. In *1st Workshop on End-User Software Engineering (WEUSE 2005) at ICSE 05*, 2005.
11. J. Ruthruff, A. Phalgune, L. Beckwith, and M. Burnett. Rewarding good behavior: End-user debugging and rewards. In *VL/HCC'04: IEEE Symposium on Visual Languages and Human-Centric Computing*, 2004.
12. J. Wielemaker. An overview of the SWI-Prolog programming environment. Technical report, University of Amsterdam, The Netherlands, 2003.

The RDF Book Mashup: From Web APIs to a Web of Data

Christian Bizer, Richard Cyganiak, and Tobias Gauß

Freie Universität Berlin
chris@bizer.de, richard@cyganiak.de, tobias.gauss@web.de

Abstract. The RDF Book Mashup demonstrates how Web 2.0 data sources like Amazon, Google and Yahoo can be integrated into the Semantic Web. Following the principles of linked data, the RDF Book Mashup makes information about books, their authors, reviews, and online bookstores available on the Semantic Web. This information can be used by RDF browsers and crawlers, and other publishers of Semantic Web data can set links to it.

1 Introduction

Major web data sources like Google, Yahoo, Amazon and eBay have started to make their data accessible to third parties through web APIs. This has inspired many interesting "mashups" which combine data from multiple sources and present it through new user interfaces. Web APIs usually consist of REST or SOAP web services that deliver XML or JSON. Scripting languages play an important role in mashup development as they often serve as the "glue" that connects data sources and UI components.

The goal of the RDF Book Mashup is to show how data from web APIs can be integrated into the Semantic Web. This has two benefits. First, it adds useful data to the Semantic Web. Second, it addresses several limitations of traditional web APIs:

- They do not work with clients that have not been designed with the specific API in mind.
- Their content cannot be accessed by search engines and other generic web agents.
- Each mashup only allows access to data from a limited number of sources chosen by the developer.

In contrast, information on the Semantic Web can be used by generic clients, including RDF browsers, RDF search engines, and web query agents.

2 Linked Data

Based on Tim Berners-Lee's *Linked Data* prinicples [1, 4], we identify a lowest common denominator for Semantic Web interoperability:

1. All items of interest, such as information resources, real-world objects, and vocabulary terms are identified by URI references [3].
2. URI references are dereferenceable; an application can look up a URI over the HTTP protocol and retrieve an RDF description of the identified item.
3. Descriptions are provided using the RDF/XML syntax.
4. Every RDF triple is conceived as a hyperlink that links to related information from the same or a different source and can be followed by Semantic Web agents.

These principles are sufficient to create a *Web of Data* in which anyone can publish information, link to existing information, follow links to related information, and consume and aggregate information without necessarily having to fully understand its schema nor to learn how to use a proprietary Web API.

3 How to Wrap Web APIs

The RDF Book Mashup applies these principles to web APIs that supply data about books. It assigns URIs to books, authors, reviews, online bookstores and purchase offers. When a URI is dereferenced, the mashup queries the Amazon API[1] for information about the book, and the Google Base data API[2] for purchase offers from different bookstores. An RDF description is assembled from the query results and returned to the client in RDF/XML syntax. The description uses a mix of RDFS vocabularies, including Dublin Core[3] and FOAF[4]. We introduced new vocabulary terms where we could not reuse existing terms from well-established vacabularies. Figure 1 gives an overview about the architecture of the RDF Book Mashup.

Figure 2 shows the description of the book identified by this URI:

`<http://www4.wiwiss.fu-berlin.de/bookmashup/books/006251587X>`

Figure 3 shows the description of its author, who is identified by

`<http://www4.wiwiss.fu-berlin.de/bookmashup/persons/Tim+Berners-Lee>`

Both URIs are linked to each other by a `dc:creator` triple. The descriptions contain links to further URIs for reviews and purchase offers, which in turn can be dereferenced to yield RDF descriptions.

[1] `http://aws.amazon.com/`
[2] `http://code.google.com/apis/base/`
[3] `http://dublincore.org/documents/dces/`
[4] `http://xmlns.com/foaf/0.1/`

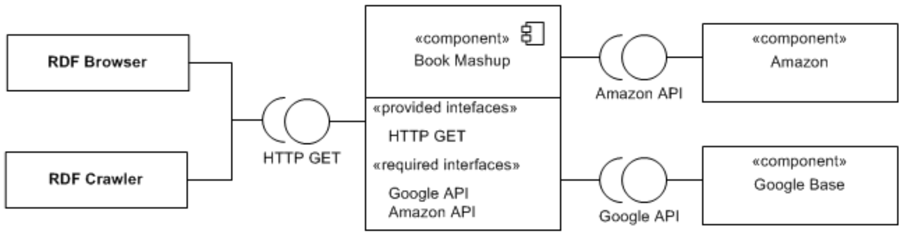


Fig. 1. Architecture of the RDF Book Mashup

4 Implementation

The RDF Book Mashup is implemented in about 300 lines of PHP code. Whenever the script gets a lookup call for a URI, it decodes the ISBN number or author name from the URI and uses them to query the origin data sources. The resulting XML responses are parsed using PHP's SimpleXML functions and turned into an RDF model which is serialized to RDF/XML using the RDF API for PHP (RAP)[5].

The services use different formats for queries and responses: The Amazon Web Service is a REST-based API that encodes requests in a URL string and answers simple XML. The Google Base data API uses a proprietary query language and delivers search results as Atom feeds. Both have in common that requests are encoded into a simple URL string, and responses are delivered as XML, which means that processing can be done easily with built-in PHP functions.

To comply with web architecture[6], the mashup uses HTTP 303 redirects to point clients from the URI for a book or person (e.g. `<books/006251587X>`) to an RDF document describing the item (e.g. `<doc/books/006251587X>`).

The generated RDF contains metadata about the document itself, including links to the license terms under which the data is published.

5 RDF Links to and from other Data Sources

RDF is the obvious technology to interlink data from various data sources. In the following, we describe how the RDF Book Mashup is interlinked with various other Semantic Web data sources.

[5] `http://sites.wiwiss.fu-berlin.de/suhl/bizer/rdfapi/`
[6] `http://norman.walsh.name/2005/06/19/httpRange-14`

```
<scom:Book rdf:about="/bookmashup/books/006251587X">
   <dc:title>Weaving the Web</dc:title>
   <dc:creator rdf:resource="/bookmashup/persons/Tim+Berners-Lee"/>
   <dc:date>2000-11-01</dc:date>
   <dc:format>Paperback</dc:format>
   <dc:identifier rdf:resource="urn:ISBN:006251587X"/>
   <dc:publisher>Collins</dc:publisher>
   <foaf:thumbnail rdf:resource=
         "http://ec2.images-amazon.com/images/P/006251587X.jpg"/>
   <rev:hasReview rdf:resource=
         "/bookmashup/reviews/006251587X_EditorialReview1"/>
   <rev:hasReview rdf:resource=
         "/bookmashup/reviews/006251587X_EditorialReview2"/>
   <scom:hasOffer rdf:resource=
         "/bookmashup/offers/006251587XamazonOffer"/>
   <scom:hasOffer rdf:resource=
         "/bookmashup/offers/006251587XgoogleOffer10871997653977"/>
</scom:Book>
<rdf:Description rdf:about="/bookmashup/doc/books/006251587X">
   <dc:license rdf:resource="http://www.amazon.com/AWS-License"/>
   <dc:license rdf:resource="http://www.google.com/terms_of_service.html"/>
</rdf:Description>
```

Fig. 2. RDF description of a book (edited)

5.1 Outgoing RDF Links

Another publicly available source of bibliographic data is the DBLP database[7]. It contains journal articles and conference papers. In a previous project, we have published this database[8] on the Semantic Web using D2R Server [2]. The RDF Book Mashup automatically generates outgoing `owl:sameAs` links from book authors to paper authors in the DBLP database. An RDF crawler that aggregates RDF statements from both sources can conclude that both URIs refer to the same person and thus allow queries over merged data from both sources.

The links are generated by asking the DBLP SPARQL endpoint for URIs identifying book authors. If the query for a `foaf:Person` with a specific name returns exactly one result, we assume (sometimes incorrectly) that they are the same person, and add the `owl:sameAs` link. An example can be seen in Figure 3.

5.2 Incoming RDF Links

To integrate the book data into the Semantic Web, we also need incoming links that enable Semantic Web clients to discover the data.

[7] `http://dblp.uni-trier.de/`
[8] `http://www4.wiwiss.fu-berlin.de/dblp/`

```
<foaf:Person rdf:about="/bookmashup/persons/Tim+Berners-Lee">
   <owl:sameAs rdf:resource=
"http://www4.wiwiss.fu-berlin.de/dblp/resource/person/100007"/>
   <foaf:name>Tim Berners-Lee</foaf:name>
</foaf:Person>
<scom:Book rdf:about="/bookmashup/books/006251587X">
   <dc:creator rdf:resource="/bookmashup/persons/Tim+Berners-Lee"/>
   <rdfs:label>Weaving the Web</rdfs:label>
</scom:Book>
<scom:Book rdf:about="/bookmashup/books/8432310409">
   <dc:creator rdf:resource="/bookmashup/persons/Tim+Berners-Lee"/>
   <rdfs:label>
      Tejiendo La Red - El Inventor del WWW Nos Descubre
   </rdfs:label>
</scom:Book>
```

Fig. 3. RDF description of an author (edited)

Another Semantic Web data source that contains information about books is DBpedia[9], a project that publishes RDF data extracted from Wikipedia. DBpedia uses the ISBN numbers that are found in Wikipedia articles to auto-generate links to the RDF Book Mashup. The DBpedia dataset currently contains about 9000 `owl:sameAs` links pointing at RDF Book Mashup URIs.

A second Semantic Web data source that auto-generates RDF links to the RDF Book Mashup is the D2R server that publishes the DBLP bibliography. The server includes a link to the RDF Book Mashup into the RDF description of each bibliography item that has a ISBN number.

The RDF Book Mashup can also be used to augment webpages. An example of a webpage that uses Book Mashup data is the Semantic Web book list [10] within the W3C ESW Wiki. The list contains about 40 manually-set links to RDF Book Mashup data about books.

A further source of incoming links are the FOAF profiles of book authors. Tim Berners-Lee and Ivan Herman are among the people who have such links in their profiles [11]. The RDF Book Mashup's website provides instructions that tell book authors how to set `dc:creator` links from their FOAF profile to their books.

6 Client Applications

Linked data can be consumed by different kinds of clients on behalf of a user. Some usage scenarios are:

[9] http://dbpedia.org/

[10] http://esw.w3.org/topic/SwBooks

[11] http://www.w3.org/People/Berners-Lee/card and http://www.ivan-herman.net/foaf.rdf

Browsing: Semantic Web browsers like Tabulator [4], Disco[12], and the OpenLink RDF Browser[13] can be used to explore and visualize Book Mashup data, and to surf into and out of the dataset.

Searching and querying: RDF crawlers can harvest the data for search engines such as SWSE[14] and Swoogle[15]. Query agents like the Semantic Web Client Library[16] can crawl the data on-the-fly to answer queries.

Augmenting web pages: Web page authors can add `<link rel="alternate">` tags[17] to web pages about books. This allows web surfers to use browsers like Piggy Bank[18] to collect book descriptions from the Web.

7 Conclusion

We have shown how to integrate information from web APIs with the Semantic Web, making them part of the Web of Data, and making them accessible to generic Semantic Web clients like RDF browsers, RDF crawlers and query agents. This has been done by wrapping several APIs into an interface that applies the Linked Data principles. The wrapper is implemented as a fairly simple PHP script. We hope that this work will inspire others to integrate further interesting data sources into the growing Semantic Web.

The source code of the RDF Book Mashup is available under GPL license. More information about the project is found at `http://sites.wiwiss.fu-berlin.de/suhl/bizer/bookmashup/`.

References

1. Tim Berners-Lee. Linked data, 2006. `http://www.w3.org/DesignIssues/LinkedData.html`.
2. Christian Bizer and Richard Cyganiak. D2r server - publishing releational databases on the semantic web. In *Poster at the 5th International Semantic Web Conference*, 2006.
3. Leo Sauermann, Richard Cyganiak, and Max Völkel. Cool URIs for the Semantic Web. Technical report, 2006.
4. Tim Berners-Lee et al. Tabulator: Exploring and analyzing linked data on the semantic web. In *Proceedings of the 3rd International Semantic Web User Interaction Workshop*, 2006. `http://swui.semanticweb.org/swui06/papers/Berners-Lee/Berners-Lee.pdf`.

[12] `http://sites.wiwiss.fu-berlin.de/suhl/bizer/ng4j/disco/`
[13] `http://demo.openlinksw.com/DAV/JS/rdfbrowser/index.html`
[14] `http://swse.deri.org/`
[15] `http://swoogle.umbc.edu/`
[16] `http://sites.wiwiss.fu-berlin.de/suhl/bizer/ng4j/semwebclient/`
[17] `http://www.w3.org/TR/rdf-syntax-grammar/\#section-rdf-in-HTML`
[18] `http://simile.mit.edu/piggy-bank/`

Rapid ontology-based Web application development with JSTL

Angel López-Cima[1], Oscar Corcho[2], Asunción Gómez-Pérez[1]

[1]OEG - Facultad de Informática. Universidad Politécnica de Madrid (UPM) Campus de Montegancedo, s/n. 28660 Boadilla del Monte. Madrid. Spain
{alopez, asun}@fi.upm.es
[2]University of Manchester. School of Computer Science. Oxford Road, Manchester, United Kingdom
Oscar.Corcho@manchester.ac.uk

Abstract. This paper presents the approach followed by the ODESeW framework for the development of ontology-based Web applications. ODESeW eases the creation of this type of applications by allowing the use of the expression language JSTL over ontology components, using a data model that reflects the knowledge representation of common ontology languages and that is implemented with Java Beans. This framework has been used for the development of a number of portals focused on the dissemination and management of R&D collaborative projects.

Introduction

Current Web applications can be designed and implemented with a wide variety of programming languages and underlying frameworks, techniques and technologies (.NET, AJAX, Java, PHP, ASP, JSP, JSTL, JDO, COM, J2EE, etc.). These technologies are widely accepted, what eases their reuse in application development, from code snippets to large applications. Besides, most of them are supported by Web authoring tools (e.g., Macromedia, FrontPage), making it easier for non-experts to create Web applications.

In the context of the Semantic Web there is already a large diversity of tools for editing and managing ontologies, annotating resources, querying and reasoning with them, etc. These tools constitute the basic building blocks of the underlying infrastructure for ontology-based application development [8]. In the past few years there have been also many efforts focused on the provision of technology for the rapid development of ontology-based Web applications. [3] describes the most relevant technologies according to a set of characteristics like their model storage, API paradigm, supported serialization formats, query languages, etc. Most of these efforts are oriented towards the easy management of RDF resources, statements and models, with a lack of approaches for the management of the most usual ontology components (classes, properties, hierarchies, and individuals). This is classified as the *ontology-centric view* in [3], and it is only supported by some of the most common Java APIs (Sesame, Jena, Protégé-OWL, etc.), which operate at a lower infrastructure level, and two other more user and Web oriented APIs, such as Spiral[1] and RAP [16], integrated with .NET and PHP respectively.

On another dimension, authoring tools are also being created for non-experts. Some of the most recent work on this side has been focused on the development of semantic wikis (e.g., [10], [14]), which allow non-experts to include ontology-based annotations (and in some cases even ontology term definitions) of the Web pages that they edit, and on the development of semantic desktops (e.g., [4], [11], [6], [1]).

In this paper we present how the ODESeW framework [10] combines both types of approaches, scripting languages for developers that want to create ontology-based Web applications and ontology-based Web authoring tools for non-experts, in a common framework. On the first aspect, ODESeW proposes the use of Java Beans for the management of ontology components and ontology-based annotations, and JSTL default and extended tags for the visualisation of ontology-based annotations. These technologies can be easily combined with other standard visualisation and navigation ones, hence improving reuse and maintainability. On the second aspect, ODESeW provides forms that are automatically generated from the underlying ontologies and allow a more structured edition of annotations.

The structure of this paper is organized as follows. Section 2 describes the ODESeW architecture. Sections 3, 4 and 5 focus on the most relevant components of this architecture together with some examples from the Person and Organization ontologies in R&D collaborative project. Section 6 provides some conclusions and future work.

[1] http://www.semanticplanet.com/library/Spiral/HomePage?from=RdfLib

The ODESeW Architecture

The ODESeW architecture, shown in Fig. **1**, is based on the Model-View-Controller design pattern [7], which is currently widely used for developing Web applications. This pattern is useful to develop applications where the same information may have several visualisations. It divides functionality among three types of objects: the model, the view and the controller.

- The model represents business data, and the business logic that governs its access and modification. In ODESeW, business data is represented with three models, coordinated by the Data Model Manager:
  - The Data Model, which contains domain information, represented by means of ontologies.
  - The User Model, which stores user profiles.
  - The External Information Gateway (EIG), which accesses external resources and annotates them with domain information.
- Views render the contents of models. They access data from the model and specify how that data should be presented. They also update data presentation when their corresponding model changes, and forward user inputs to the controller.
- The controller defines the application behaviour. It dispatches user requests (button clicks, menu selections, form input texts, etc.), also known as user gestures or actions, interpreting and mapping them into actions to be performed by the model. These actions can perform navigation of the user from one view to another and also execute business logic of the application.

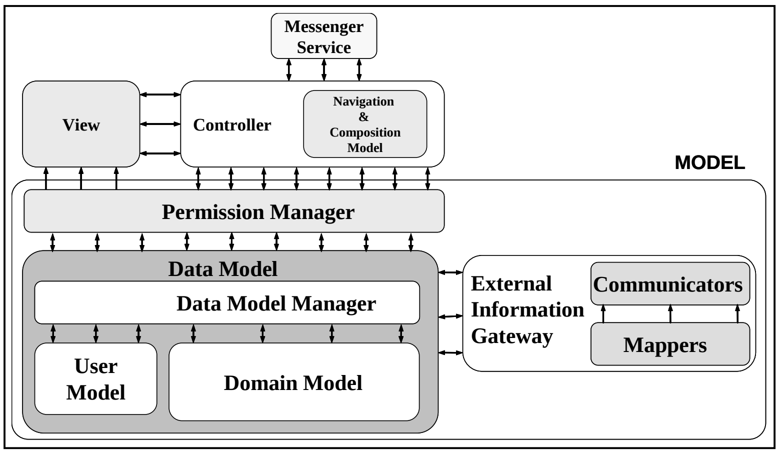


Fig. 1. ODESeW Architecture

The ODESeW architecture also includes other services, such a the Messenger Service, in charge of communicating ODESeW applications with external applications. For instance, the controller may send a message to external applications when an instance is created, updated or removed, or receive a message when an external information source changes. Another important component is the permission manager, which filters user requests and responses according to the user that is accessing the application. The description of how these components work is out of scope of this paper, and can be found in [12].

Data Model

The ODESeW data model uses a frame-based ontology representation model, based on the WebODE [2] knowledge representation ontology, which is the underlying ontology repository used by ODESeW. Fig. 2 shows the components in this model (concepts, attributes, relations, instances, etc.) and the relations between them, and Fig. 2 shows all the attributes and relations of each component of the model. This model has been implemented with Java Beans [9], what enables the use of a large amount of third party Web application development technologies.

The Data Model Manager manages connections to the ontology repositories that store the domain application and the user models. This includes starting up connections, managing their lifetime, and identifying the relevant ontologies to be used. Since the current implementation connects to WebODE, ontologies implemented in a range of languages (including RDF Schema and OWL) can be imported in the system and used.

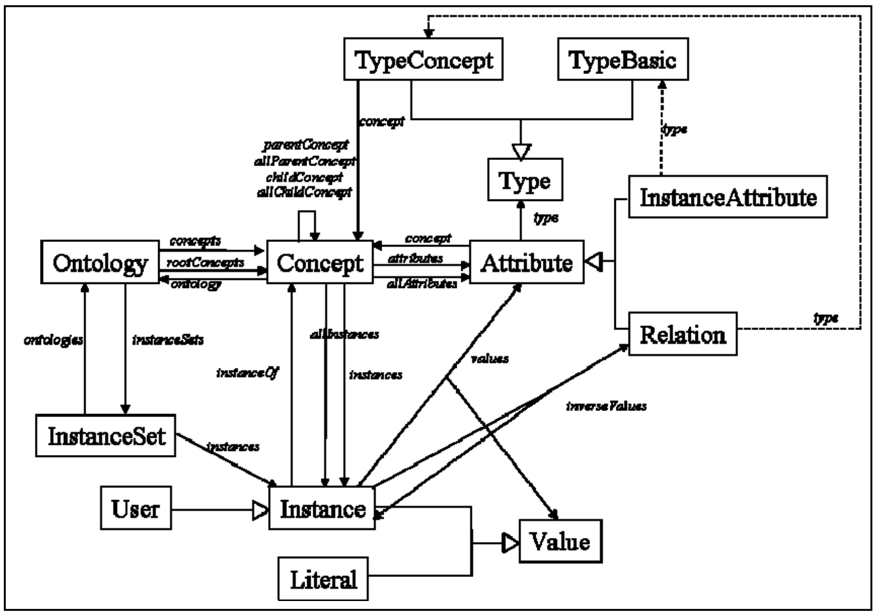


Fig. 2. ODESeW Data Model diagram

View

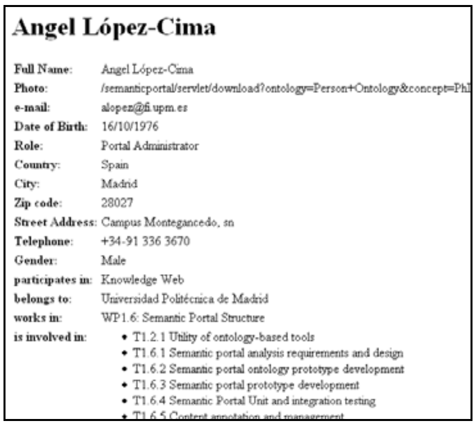
ODESeW views are implemented with XHTML, for static pages, and with JSP [15], for dynamic pages. The JSP pages use a combination of standard JSTL tags with the Expresion Language [15], and of custom tags to access information from the data model, as illustrated in Fig. 3 and Fig. 5.

The code included in Fig. 3 is included in the page *instance.jsp*, and uses JSTL to execute a set of commands that display the information about a person. The commands are executed are the following: 1) get the instance "Angel López-Cima" from the ontology "Person Ontology"; 2) print the instance name, 3) iterate for all pairs <Attribute, Value[]> of instance values; 4) set the attributes *attribute* and *values* from the pair <Attribute, Value[]>; 5) store in the attribute *multivalue* if the instance has more than one value for the attribute *att*, and in that case create an enumeration list with the following instructions; 6) print the attribute name of the pair <Attribute, Value[]>; 7) iterate for all values in the pair <Attribute, Value[]>; 8) and print a value. Fig. 4 is the page that results from the execution of the code inside this page. To display any other instance, no matter which ontology it comes from or which concept it belongs to, the web developer only needs to change the values in step 1) in Fig. 3.

```
<sew:ontology var="persOntology" name="Person Ontology">              ①
  <sew:instance var="instance" name="Angel López-Cima"/>
</sew:ontology>

<h1>${instance.name}</h1>                                              ②
  <table>
  <c:forEach var="pairAttVals" items="${instance.values}">             ③
    <c:set var="attribute" value="${pairAttVals.key}"/>                ④
    <c:set var="values" value="${pairAttVal.value}">
    <c:set var="multivalue" value="${fn:length(values)>1}"/>           ⑤
  <tr>
    <th align="left" valign="top">${attribute}:</th>                  ⑥
    <td align="left">
    <c:if test="${multivalue}"><ul></c:if>
    <c:forEach var="value" items="${values}">                          ⑦
      <c:if test="${multivalue}"><li></c:if>
      ${value}                                                         ⑧
      <c:if test="${multivalue}"></li></c:if>
    </c:forEach>
    <c:if test="${multivalue}"></ul></c:if>
    </td>
  </tr>
  </c:forEach>
  </table>
```

Fig. 3. *Instance.jsp*, a simple view of an instance.

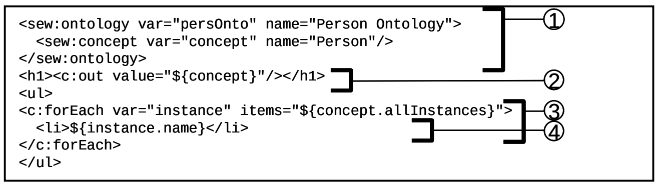


Angel López-Cima

Full Name:	Angel López-Cima
Photo:	/semanticportal/servlet/download?ontology=Person+Ontology&concept=PhI
e-mail:	alopez@fi.upm.es
Date of Birth:	16/10/1976
Role:	Portal Administrator
Country:	Spain
City:	Madrid
Zip code:	28027
Street Address:	Campus Montegancedo, sn
Telephone:	+34-91 336 3670
Gender:	Male
participates in:	Knowledge Web
belongs to:	Universidad Politécnica de Madrid
works in:	WP1.6: Semantic Portal Structure
is involved in:	• T1.2.1 Utility of ontology-based tools • T1.6.1 Semantic portal analysis requirements and design • T1.6.2 Semantic portal ontology prototype development • T1.6.3 Semantic portal prototype development • T1.6.4 Semantic Portal Unit and integration testing • T1.6.5 Content annotation and management

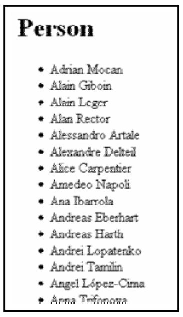
Fig. 4. Simple visualization of an instance of the concept Person.

The code included in Fig. 5 executes the following commands: 1) get the concept "Person" from the ontology "Ontology Person"; 2) print the name of the concept; 3) iterate for all direct and indirect instances of the concept; 4) print the name of an instance. Fig. 6 is the page that results from its execution. To display any other concept, the web developer only needs to change the value in 1) in Fig. 5.



```
<sew:ontology var="persOnto" name="Person Ontology">
  <sew:concept var="concept" name="Person"/>
</sew:ontology>
<h1><c:out value="${concept}"/></h1>
<ul>
<c:forEach var="instance" items="${concept.allInstances}">
  <li>${instance.name}</li>
</c:forEach>
</ul>
```

Fig. 5. *Concept.jsp*, a simple view of a concept



Person

- Adrian Mocan
- Alain Giboin
- Alain Leger
- Alan Rector
- Alessandro Artale
- Alexandre Delteil
- Alice Carpentier
- Amedeo Napoli
- Ana Ibarrola
- Andreas Eberhart
- Andreas Harth
- Andrei Lopatenko
- Andrei Tamilin
- Angel López-Cima
- Anna Trifonova

Fig. 6. Simple visualization of the list of instances of the concept Person.

Controller

The ODESeW Controller receives user requests, which contain the actions to be performed, and completes or checks requests with the information model in the Data Model (including both the domain and the user models). Then it reads and executes the navigation and composition model, described below, and returns the next view that should be rendered for the user.

We will describe first the ODESeW Navigation and Composition Model, and then the steps followed by the Controller to execute actions.

The Navigation and Composition Models

The **navigation model** represents the navigation of a user through the application. This model is explicitly separated from the design of views so that changes in the navigation do not affect the implementation of views. Besides, it allows representing declaratively the navigation of a user, enabling in this way an easy study of the behaviours of the user of an application.

The navigation model is a directed named graph in which nodes represent views and edges represent navigation actions from one view to another.

- Nodes have 2 attributes: "*precondition*" and "*view URL*". The first one specifies preconditions to allow the execution of a view and the second one specifies the location of the view. If the precondition is empty, it is considered to be true. If the *viewURL* attribute is empty this means that the view is abstract. That is, it is a view that cannot be rendered directly and has to be specialised by other views so as to be used by the Controller.
- Edges identify actions that can be performed from a view. Besides redirecting users from a view to another, edges are attached to a task execution: instance edition, instance removal, message sending, etc. An edge may not have an origin node, that is, the action represented by this edge can be executed from all pages, represented or not in the navigation model. Besides, edges can be concatenated to perform different tasks in a navigation step.

The navigation model also allows describing specialisation/generalisation relations between two views (defined with the subclass-of relationship). A view is a specialisation of another if it visualises the same content as its parent view but providing more specific visualisation items. For instance, a default view may be used to render any instance and for other more specific instances, such as instances of persons, instances of publications, etc., other more specific views can be created.

Fig. 7 shows an example of a navigation model with 9 views defined and several types of actions and specialisation/generalisation relations defined between them.

The **composition model** is similar to the navigation model, though its rationale is different: it allows including a set of views inside another and is normally used when complex sets of information have to be presented at once.

One common example of the use of the composition model is for displaying a default view that render attribute values and for other more specific types of values, such as e-mail addresses, URLs, image files, sound files, video files, etc., other more specific views can be created.

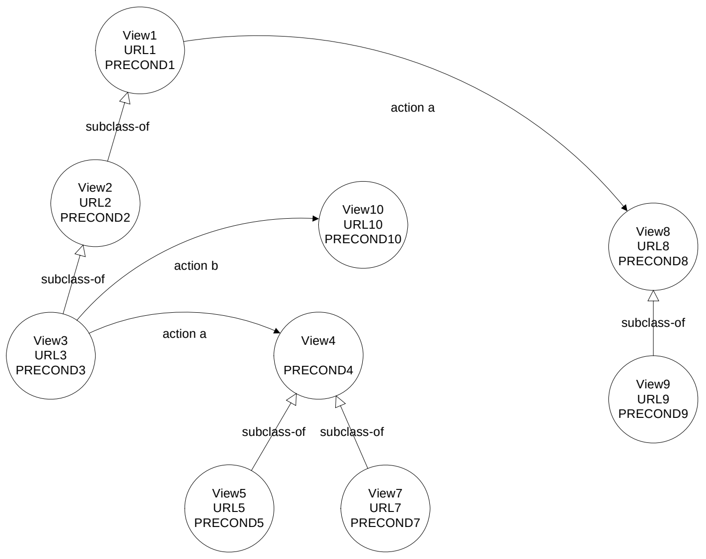


Fig. 11. Example of a navigation model

The elements used in the composition model are the same as those for the navigation model: views are represented as nodes, with the attributes "*precondition*" and "*view URL*"; views can be specialized with other views; and actions are represented as edges. The only constraint is the type of actions that can be represented in this model, which only consists in the action of inclusion of a view inside another.

Controller Execution

Actions received by the Controller contain two elements: task and control flow operation. The task is the specific operation to be performed, while the control flow operation specifies what to do after the execution of the task.

Developers can use any of the default tasks provided by ODESeW or create new ones, either from scratch or by reusing and extending any of the default ones. The following default tasks are available: *sewView*. It renders the view specified in the user request by redirecting users to it; *sewRemove*. It deletes the set of concept and relation instances specified in the user request; *sewEdit*. It updates or creates the set of concept and relation instances specified in the user request; *sewSearch*. It searches for a set of concept and relation instances that satisfy the query; *sewRouter*. It is used to execute another action from a list specified in the user request. These actions have a guard condition, and the sewRouter task selects the first one whose guard condition is satisfied; *sewLogin*. It authenticates a user and loads his/her profile in the user session.

With respect to control flow operations, there are four available: *Forward*: the user request is concatenated to another action or view; *Redirect*: the user request ends and a new user request starts. This new request consists in showing another view or performing another action; *Include*: the execution of a new action or view is included in the original view or action, so that it will be performed later; *Empty*: the execution ends without any more control flow actions. In fact, a view is actually defined as a rendering action, optionally followed by other additional include actions, and which has an empty control flow at the end.

When a user requests an action from a view, the Controller executes the navigation model, following these steps:

1. Identify the view from which the user request is originated, and find it in the navigation model.
2. Find the requested action in the source view. The action can be defined explicitly in the source view or in any of its ancestor views.
3. Select the target view for the requested action. In the navigation model, an action applied to a specific view may have several target views, and at least one of them has to be selected. To perform this selection, the Controller verifies whether the precondition of any of the target views specified in the

action is satisfied given the request parameters. If no precondition is satisfied, an exception raises and the error is reported to the portal administrator.

4. Find whether any of the specialisations of the selected target view is also valid. Once the controller found a valid candidate view, it will try to find another one among its specialisations. To do this, the Controller checks the preconditions of the view specialisations. If any of them is satisfied, that view is a new valid candidate view and the Controller repeats this step with its children views, until a valid view does not have more specialisations or none of the preconditions of its specialisations are satisfied. The last valid candidate view is the final target view.

Let us see an example based on the navigation model presented in Fig. 11. Let us assume that the user requests the **action a** from the view **View3**, and that the parameters of the request satisfy the preconditions **Precondition4**, **Precondition8** and **Precondition9** and do not satisfy the preconditions **Precondition5** and **Precondition7**.

First, the Controller finds the source view (**View3**). Taking into account that the user wants to perform **action a**, the possible candidate views are the **View4** and **View8**.

The first candidate to be checked is **View4**. However, **View4** is abstract, so the Controller has to check the preconditions of its specialisations (**View5** and **View7**). Neither of them satisfies the preconditions, so View4 nor its specialisations are valid target views. Hence, **View4** is discarded by the Controller and the next candidate view (**View8**) is analysed. The Precondition8 is satisfied, hence the **View8** is a valid candidate view. Then, the Controller starts looking for its specialisations (**View9**). The precondition of **View9** is also satisfied and, since **View9** does not have specialisations, the final target view for the execution of action from **View3** is the **View9** (see Fig. 12).

Both models, the navigation and composition, can be included and executed under the technologies Struts[2] or JSF [5].

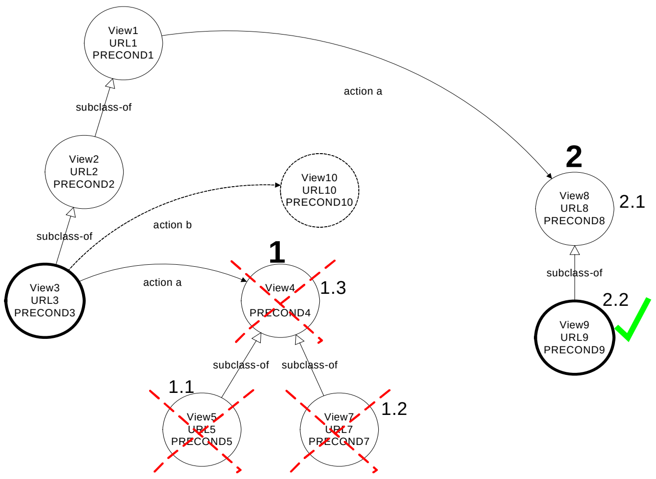


Fig. 12. Example of a navigation model execution.

Examples of composition and navigation

Now we show how we can improve the visualisation of the examples presented in Fig. 3 and in Fig. 5 by including specialised visualisations for images, emails and relations to other instances, using the composition and navigation models described above.

Using the Composition Model, we include a hierarchy of visualization of values where each node of the hierarchy is a type of attribute and a visualization of instances inside the visualization of a concept that lists all instances in a concept Fig. 15 shows a Navigation Model for visualizing values in an instance and instances in the list of instances from a concept and Fig. 16 it shows the JSP code that visualized each type of attributes and the instance.

Executing the page presented in the Fig. 3, but substituting the line marked with number 8) with the following line:

```
<sew:out action=“includeView” value=“${value}” attribute=“${att}”/>
```

and Fig. 5, but substituting the line marked with number 4) with the following line:

```
<sew:out action=“includeView” value=“${instance}”/>
```

the portal displays to the user the visualization presented in Fig. 17 in the left side.

[2] http://struts.apache.org/index.html

When the web server executes the page *instance.jsp* and finds the tag `<sew:out>`, the ODESeW controller looks for the best visualization that matches the original view *instance.jsp*, executing the action *includeView* with the parameters depending on the attribute of the value and the actual value to be displayed. And when the web server executes the page *concept.jsp* and finds the tag `<sew:out>`, the ODESeW controller includes *refInstance.jsp* inside the page *concept.jsp*.

In this way, web developers delegate to the Composition Model how to visualize each type of information, saving a lot of time that would be needed to create a large set of if-then-else or switch-case commands in all dynamic pages that are used to visualize, in a specific format, attribute values. In the last example, if we remove the lines marked by 1) in Fig. 3 and in Fig. 5 , the view is fully reusable for displaying any instance and concept, and if the user click on any destination instance, ODESeW executes the same *instance.jsp* but displaying another instance, as shown in Fig. 17.

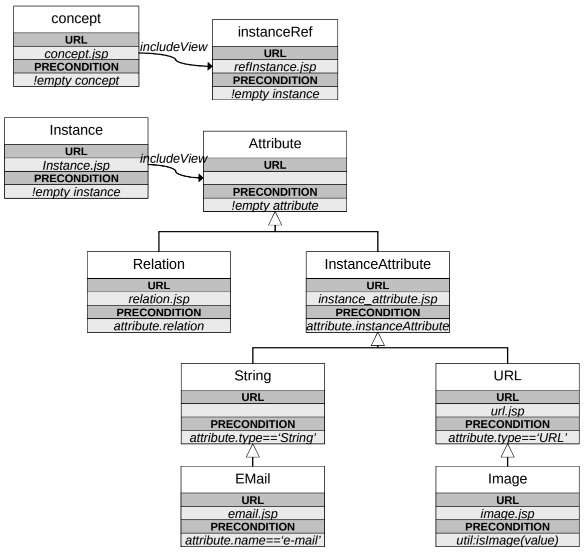


Fig. 15. Composition Model for value visualizations

relation.jsp

```
<sew:url var="link" value="instance.jsp">
  <sew:param name="instance" value="${value}">
</sew:url>
<a href="${link}">${value.name}</a>
```

instance_attribute.jsp

```
${value}
```

url.jsp

```
<a href="${value}">${value}</a>
```

image.jsp

```
<img src="${value}" alt="${attribute.name}" />
```

email.jsp

```
<a href="mailto:${value}">instance.name</a>
```

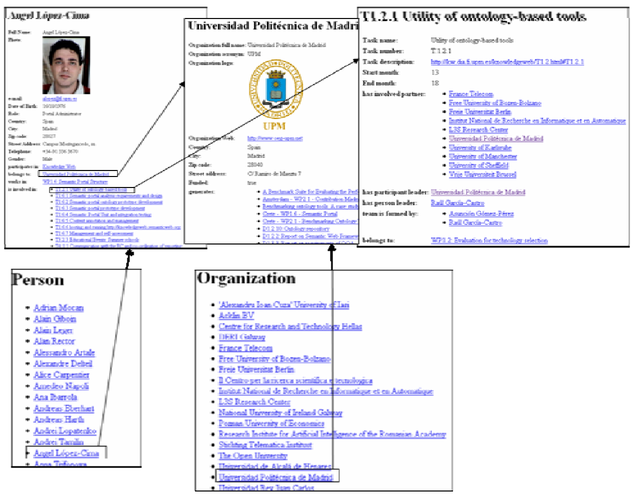
refInstance.jsp

```
<sew:url var="link" value="instance.jsp">
  <sew:param name="instance" value="${instance}">
</sew:url>
<a href="${link}">${instance.name}</a>
```

Fig. 16. Value visualizations

Now, the *instance.jsp* and *concept.jsp* pages can be used to create generic views for displaying any instance and concept in the Data Model. However, in most of the cases, the generic view is not what the web developer wants to display in all cases. ODESeW allows web designers to create specific visualizations for different types of information in the Data Model in the same way as it is done with the Composition Model, but using the Navigation Model instead.

In the Knowledge Web portal, we set a navigation model with specific views for displaying persons involved in the project, organization partners in the project, instances of persons and instances of organizations. The resulting navigation and visualization is presented in Fig. 18 and the result of their execution is shown in Fig. 19.



Ángel López-Cima

Universidad Politécnica de Madrid

T1.2.1 Utility of ontology-based tools

Person

- Adrian Mocan
- Alain Giboin
- Alain Leger
- Alan Rector
- Alessandro Artale
- Alexandre Delteil
- Alice Carpentier
- Amedeo Napoli
- Ana Iberrola
- Andreas Eberhart
- Andreas Harth
- Andrei Lopatenko
- Andrei Tamilin
- Angel López-Cima
- Anna Trifonova

Organization

- 'Alexandru Ioan Cuza' University of Iasi
- Aidin BV
- Centre for Research and Technology Hellas
- DERI Galway
- France Telecom
- Free University of Bozen-Bolzano
- Freie Universitat Berlin
- Il Centro per la ricerca scientifica e tecnologica
- Institut National de Recherche en Informatique et en Automatique
- L3S Research Center
- National University of Ireland Galway
- Poznan University of Economics
- Research Institute for Artificial Intelligence of the Romanian Academy
- Stichting Telematica Instituut
- The Open University
- Universidad de Alcalá de Henares
- Universidad Politécnica de Madrid
- Universidad Rey Juan Carlos

Fig. 17. Visualization of different instances with the same *instance.jsp* and *concept.jsp*

To execute the navigation through the global navigation action *viewTerm*, we substitute the value of the attribute *value* of the tag `<sew:out>` of the files *relation.jsp* and *refInstance.jsp* shown in Fig. 16 from the value *instance.jsp* to the value */sew/viewTerm* (e.g. `<sew:out var="link" value="/sew/viewTerm">`...). In this way, when the portal receives a request for executing the action *viewTerm*, and the parameters contain a class or subclass of *Organization*, then it shows the page */concept/organization.jsp*. If the parameters contain a class or subclass of *Person*, then it shows the page */concept/person.jsp*. If the parameters contain any other class, then it shows the generic concept visualisation page *concept.jsp*. Similarly this will happen with instances of *Organization*, *Person*, etc.

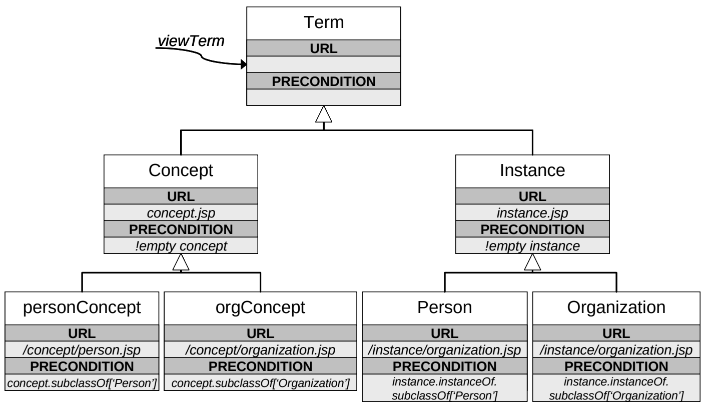


Fig. 18. Navigation Model

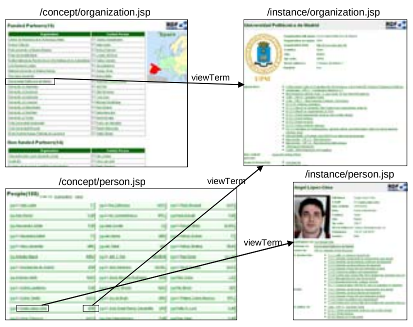


Fig. 19. Visualization of different objects in the Data Model.

Instance edition

ODESeW also provides a set of tags that help to create specific forms for editing instances, together with actions to store those modifications inside the Data Model. The set of custom JSTL tags provided by ODESeW are extensions of the well-known HTML tags for creating form objects, as shown in Table 2.

In the case that the form is editing an instance or a group of instances, all the ODESeW tags of the HTML form object have 3 new attributes: *instance* (here the web designer specifies which instance is being edited); *attribute* (here the web designer specifies which instance attribute is being edited); and *mode* (here the web designer specifies whether the value input by the user will be appended to the existing values or will replace the old values stored in the ontology). Besides, in this case, the attribute *value* is used to specify a destination instance of a relation.

Table 2. ODESeW tag for HTML forms

<sew:form>	Specification of a form and how information will be submitted.	
	name, action, enctype, method, target, all event triggers	Standard attributes from the <form> tag
<sew:input> <sew:textarea> <sew:button> <sew:select>	Object form: text input, checkbox, option, button, text area, a combo list and a multivalue list	
	accept, accesskey, align, alt, border, checked, cols, datafld, datasrc, disabled, id, ismap, maxlength, name, readonly, rows, size, src, style, tabindex, type, usmap, all event triggers	Standard attributes from the <input>, <textarea>, <button> and <select> tag.
	value	Standard attribute, but it can contain any term of the Data Model.
	instance	Sets the instance that is being edited
	attribute	Sets the instance attribute being edited
	mode	Sets whether the value in this tag will be appended [`append`] in the instance or will be used to replace the old values [`update`].
<sew:option>	Option in a combo box	
	disabled, label, selected	Standard attributes from <option> tag
	value	Standard attribute, but it can contain any term of the Data Model.

In the case that the form is requesting an attribute that contains a term of the Data Model (ontology, concept, attribute or instance), the attribute *value* can contain one of these terms, which is passed as a parameter to the action of the form.

These tags include a set of *javascript* functions that are in charge of generating hidden controls in the form that are used in an editing instance action from ODESeW to collect the values of the instances entered in the form. These *javascript* functions do not generate noise behaviour in the generated HTML and allow web designers to include their own *javascript* functions as usual.

Fig. 22 shows the composition model for editing instances, and Fig. 24 and Fig. 25 show that forms are specified very similarly to how they are done in the normal visualization shown in Fig. 19, and that they can generate a generic form for any instance.

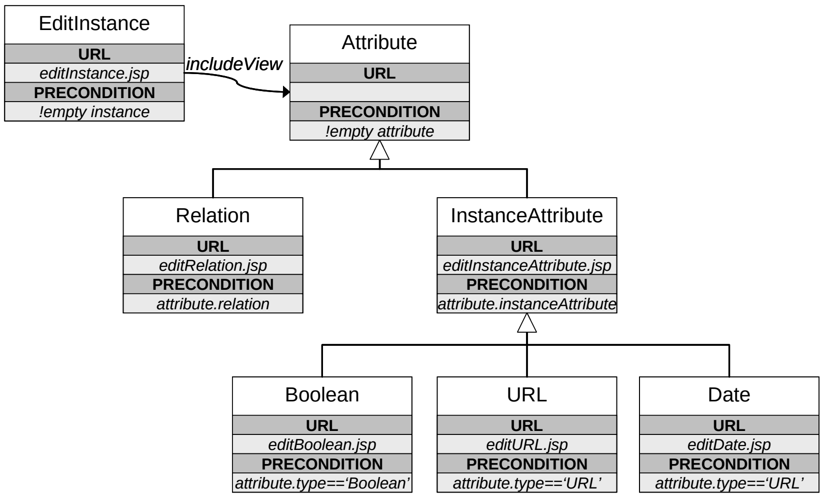
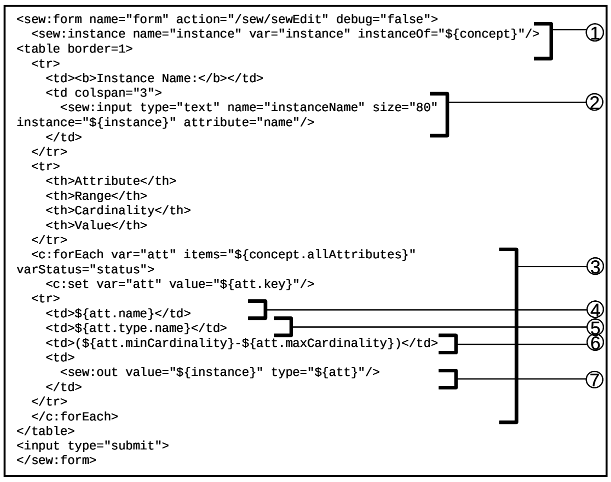


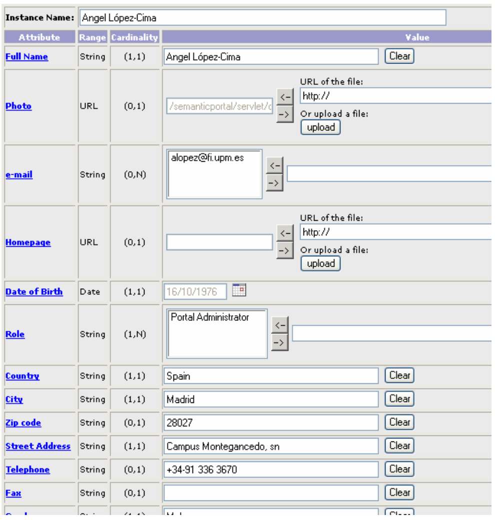
Fig. 22. Composition Model for editing instances

The code include in the Fig. 24 executes the following commands: 1) sets the parameter "*instance*" of the form as a new variable of the form with the name "*instance*" and this new instance is an instance of the concept represented by the variable "*concept*"; 2) prints a text field in the form asking for the name of the instance represented in the variable "*instance*"; 3) iterates among all attributes of the concept of the instance; 4) prints the name of the attribute; 5) prints the name of the type of the attribute; 6) prints the minimum and the maximum cardinality of the attribute; 7) calls the *includeView* action specifying the instance and the attribute to include an HTML object form to edit values for that attribute in that instance. The command in 7) calls the composition model shown in Fig. 22 iteratively among all attributes of the concept of the instance and presents as a result the visualization shown in Fig. 25.



```
<sew:form name="form" action="/sew/sewEdit" debug="false">
  <sew:instance name="instance" var="instance" instanceOf="${concept}"/>
<table border=1>
  <tr>
    <td><b>Instance Name:</b></td>
    <td colspan="3">
      <sew:input type="text" name="instanceName" size="80"
instance="${instance}" attribute="name"/>
    </td>
  </tr>
  <tr>
    <th>Attribute</th>
    <th>Range</th>
    <th>Cardinality</th>
    <th>Value</th>
  </tr>
  <c:forEach var="att" items="${concept.allAttributes}"
varStatus="status">
    <c:set var="att" value="${att.key}"/>
  <tr>
    <td>${att.name}</td>
    <td>${att.type.name}</td>
    <td>(${att.minCardinality}-${att.maxCardinality})</td>
    <td>
      <sew:out value="${instance}" type="${att}"/>
    </td>
  </tr>
  </c:forEach>
</table>
<input type="submit">
</sew:form>
```

Fig. 24. editInstance.jsp for editing instances.



Instance Name:	Angel López-Cima		
Attribute	**Range**	**Cardinality**	**Value**
Full Name	String	(1,1)	Angel López-Cima Clear
Photo	URL	(0,1)	/semanticportal/servlet/c <- -> URL of the file: http:// Or upload a file: upload
e-mail	String	(0,N)	alopez@fi.upm.es <- ->
Homepage	URL	(0,1)	<- -> URL of the file: http:// Or upload a file: upload
Date of Birth	Date	(1,1)	16/10/1976
Role	String	(1,N)	Portal Administrator <- ->
Country	String	(1,1)	Spain Clear
City	String	(1,1)	Madrid Clear
Zip code	String	(0,1)	28027 Clear
Street Address	String	(1,1)	Campus Montegancedo, sn Clear
Telephone	String	(0,1)	+34-91 336 3670 Clear
Fax	String	(0,1)	Clear

Fig. 25. Visualization of editInstance.jsp

Conclusions and future work

In this paper we have presented how the ODESeW framework allows the rapid development of ontology-based Web applications using standard Web development technologies, such as Java Beans, custom JSTL tags and the Expression Language. This allows developers that are familiar with those technologies to access ontology-based information to generate ontology-based applications easily. The main advantages are on the visualisation of ontology components (including instances), on the advanced navigation and composition models, and on the edition of instances using standard HTML form edition technologies.

This framework has been used to generate several Semantic Web portals: Knowledge Web (EU Network of Excellence) at http://knowledgeweb.semanticweb.org; OntoGrid (EU project portal) at http://www.ontogrid.eu; NeOn (EU project portal) at http://droz.dia.fi.upm.es/neon; Ontology Engineering Group (research group portal) at http://www.oeg-upm.net; and Red Temática en Web Semántica (Spanish Network) at http://www.redwebsemantica.es.

Most of the future work on this framework will be devoted to the development of the External Information Gateway (EIG). This component accesses external resources, annotates them with the domain ontologies in the ODESeW data model and gives access to these annotated resources as if they were part of the internal ODESeW data model. The specification and design of this component are already finished, and the implementation is currently in progress.

Acknowledgements

This work has been supported by the EU IST Network of Excellence Knowledge Web (FP6-507482).

References

[1] A. Cheyer, J. Park, R. Giuli. “IRIS: Integrate. Relate. Infer. Share”. ISWC Workshop on Semantic Desktop. 2005.

[2] JC. Arpírez, O. Corcho, M. Fernández-López, A. Gómez-Pérez. “WebODE in a nutshell”. AI Magazine 24(3):37-48. Fall 2003

[3] C. Bizer, D. Westphal. “Developers Guide to Semantic Web Toolkits for different Programming Languages”. http://sites.wiwiss.fu-berlin.de/suhl/bizer/toolkits/. Last updated: January 2007.

[4] D. Karger, K. Bakshi, D. Huynh, D. Quan, V. Sinha. “Haystack: A General Purpose Information Management Tool for End Users of Semistructured Data”. CIDR 2005

[5] E. Burns, R. Kitain. “JavaServer Faces”. JSR-000252. http://jcp.org/en/jsr/detail?id=252

[6] G. Tummarello, C. Morbidoni, M. Nucci, “Enabling Semantic Web communities with DBin: an overview”, Proceedings of the Fifth International Semantic Web Conference ISWC 2006, November 2006, Athens, GA, USA

[7] E. Gamma, R. Helm, J. Vlissides, R. Jhonson. “Design Patterns: Elements of Reusable Object-Oriented Software”. Boston: Addison-Wesley, 1995.

[8] R. García-Castro, MC Suárez-Figueroa, A. Gómez-Pérez, D. Maynard, S. Costache, R. Palma, J. Euzenat, F. Lécué, A. Léger, T. Vitvar, M. Zaremba, D. Zyskowski, M. Kaczmarek, M. Dzbor, J. Hartmann, S. Dasiopoulou. “D1.2.4 Architecture of the Semantic Web Framework”. Knowledge Web technical report, December 2006.

[9] G. Hamilton. “JavaBeans v1.01”. 1997 http://java.sun.com/products/javabeans/

[10] J. Fischer, Z. Gantner, S. Rendle, M. Stritt, L. Schmidt-Thieme. “Ideas and Improvements for Semantic Wikis”. 3rd European Semantic Web Conference (ESWC 2006), Budva, Montenegro.

[11] L. Sauermann. “The Gnowsis Semantic Desktop for Information Integration”. IOA Workshop of the WM2005 Conference. 2005

[12] A. López-Cima, O. Corcho, A. Gómez-Pérez.. “A platform for the development of Semantic Web portals”. In: Proceedings of the 6th International Conference on Web Engi-neering (ICWE2006). Stanford, July 2006.

[13] A. López-Cima, O. Corcho, MC. Suárez-Figueroa, A. Gómez-Pérez. “The ODESeW platform as a tool for managing EU projects: the KnowledgeWeb case study”. In: Proceedings of the 15th International Conference on Knowledge Engineering and Knowledge Management Managing Knowledge in a World of Networks (EKAW2006). Podebrady, October 2006.

[14] M. Krötzsch, D. Vrandečić, M. Völkel. “Wikipedia and the Semantic Web - The Missing Links”. Proceedings of the WikiMania2005.

[15] P. Delisle, J. Luehe, M. Roth. “JavaServer Pages”. JSR-000245. http://jcp.org/en/jsr/detail?id=245

[16] R. Oldakowski, C. Bizer, D. Westphal. „RAP: RDF API for PHP”. ESWC2005 workshop on Scripting for the Semantic Web (SFSW2005). Heraklion, Crete, May 2005.

Leveraging existing Web frameworks for a SIOC explorer to browse online social communities

Benjamin Heitmann and Eyal Oren

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

Abstract. Since online Semantic Web applications are based on existing Web infrastructure, developing these applications could leverage experiences with and infrastructure of existing frameworks. These frameworks need to be extended to deal with the different nature of Semantic Web data. We introduce several extensions to the Ruby on Rails Web development framework to support Semantic Web application development, and demonstrate them by developing a SIOC explorer. This online application integrates information from heterogeneous communities, allowing users to explore this information and find relevant posts and topics across these sites without the need to manually visit the different sites.

1 Introduction

Today's World Wide Web is undergoing a transition towards tomorrow's Semantic Web. Applications on the Semantic Web can provide benefits to their users by integrating data from many different sources. One challenge that developers encounter is the lack of support for developing such Semantic Web applications, compared to the many frameworks that exist for creating "ordinary" Web applications.

One option for developers, is to implement the Semantic Web elements of their application manually and to use an existing Web application framework for creating the Web interface. But as we will explain, this option is not optimal given the fundamental differences between Semantic Web data and traditional relational data, restricting the parts of the framework that can be reused.

Instead, this paper reports on the extension and modification of an existing Web development framework with components for consuming and processing Semantic Web data. This approach leverages the existing Web development infrastructure while also catering for the requirements of the Semantic Web.

We demonstrate this framework through the development of a browser for online social communities using the SIOC vocabulary. The browser allows the end-user to explore integrated information from various community sites (forums, mailing lists, weblogs) which would otherwise need to be visited separately. The aggregated view can be filtered using the rich post meta-data and allows users to discover people's contributions or active topics across sites.

1.1 Contribution

Our framework consists of several components that extend the Ruby on Rails Web application framework. Rails is a mature framework for developing Web applications, with many focused libraries and plugins and a large and lively developer's community. The first component is our ActiveRDF library, a high-level RDF API, that uses Ruby's dynamic meta-programming features to overcome several fundamental mismatches between object-oriented data and the Semantic Web (1). The second component is our BrowseRDF library for faceted navigation of large datasets. The third component is a SIOC crawler which integrates several command-line tools to form a "Semantic Web enabled processing pipeline" for fetching SIOC data. We combine these three components into our SIOC explorer prototype.

1.2 Outline

The rest of the paper is structured as follows: Section 2 introduces our motivating scenario: integrating data from various online communities. Section 3 discusses the relation between developing "traditional" Web applications and Semantic Web applications, and describes how to leverage existing development infrastructure. Section 4 then introduces our approach to Semantic Web development, consisting of three extensions to the Ruby on Rails framework, and demonstrates this approach in our SIOC browser prototype.

2 Scenario: integrating online social communities

As an example, we consider the development of an application for collecting information from online social communities. "Online communities" is a generic term for community sites such as forums, weblogs, mailing lists or IRC channels. Some of these channels (such as forums or bulletin boards) are more centralised, others (weblogs, IRC channels) are more decentralised and disparate. But from an abstract perspective all such communities are relatively similar: they allow users to group themselves online and exchange and discuss about their particular topics of interest.

Often, discussions range over several of these communication channels. For example, to solve an installation problem of a wireless card in the Ubuntu Linux distribution, a user should search the Ubuntu community forums for some helpful advice but also look on weblogs and the ubuntu-users mailing list. Currently, users have to browse these communication channels manually and repeat their query in various different systems: the forum software, the mailing list online archives, a weblog search engine, etc. For the end-user, it would be convenient if all these community sites were aggregated in a single place, allowing him to search for solutions to his problem in only one system.

Integrating such data currently involves: (i) collecting the heterogeneous data, i.e. crawling or"screen-scraping" and then parsing various formats and websites such as RSS, Atom, (X)HTML websites, e-mail archives, IRC chatlogs; (ii) integrating these sources into a unified structure and format; (iii) consolidation of resources and concepts such as users and topics mentioned on different sites but with different identifiers (e.g. usernames or email addresses).

3 Application development from the Web to the Semantic Web

The World Wide Web is described by (2), as a combination of Hypertext and the Internet. Documents can link to other documents or parts of them, independent of the document location. This enabled the creation of interlinked webs of documents without central control or a central repository.

The quick adoption rate of this new paradigm of information sharing resulted in a demand for making dynamic content available. New systems were created, which could generate documents on behalf of user interaction and which could constantly incorporate new information into the created documents.

Tim Berners-Lee(2) not only envisioned the World Wide Web, he also outlined the vision of the Semantic Web. The Semantic Web was intended as a combination of knowledge representation using semantic networks and the Internet. Instead of having isolated repositories of knowledge, each with their own concepts and semantics, on the Semantic Web common concepts and their semantics could be shared and linked. This would allow knowledge repositories to link to each other, forming an interlinked web of data without central control or a central repository.

As standards emerge for such a Semantic Web, static but linked knowledge repositories are now possible. But, similarly to the situation on the "ordinary" web, a demand will arise for combining and processing data based on user interaction and for dynamically incorporating new data into the knowledge repositories to reflect changes. The resulting knowledge repositories will be of a dynamic nature, continuously integrating new data from various heterogeneous data sources.

3.1 The relationship between Web and Semantic Web applications

Web applications are part of the World Wide Web if they can be accessed over the Web: (i) they expose their functionality through URLs which provide access to the web application, (ii) these URLs can be used to access the human usable interface, and (iii) these URLs can also be used to access the machine processable interface, allowing integration between web applications.

Semantic Web applications are part of the Semantic Web if they can (i) consume, (ii) process, and (iii) optionally publish semantic web data. Accessed data can be the output from another Semantic Web application. Published data can in turn be used as input for another Semantic Web application.

Semantic Web applications can, additionally, be part of the World Wide Web, if they expose themselves on the Web. This class of Semantic Web applications build on existing Web infrastructure. Developing these class of applications therefore can leverage the existing frameworks for developing Web applications, as long as we can make the necessary adjustments to overcome the differences between traditional, centralised, relational data and the upcoming, decentralised, graph-based Semantic Web data.

3.2 Decentralised data on the Semantic Web

The SIOC[1] initiative aims to ease the integration of online social community information (3). SIOC provides on the one hand a unified ontology to describe such online communities, and secondly, several exporters which translate community information from weblogs, forums, and mailing lists into RDF using the SIOC vocabulary.

Semantic Web data is expressed using RDF[2], a graph-based representation language. Statements in RDF are triples consisting of a subject, a predicate and an object which assert that the subject has a property with some value. RDF Schema[3] is the schema language for the Semantic Web, allowing the description of vocabularies in terms of classes and properties.

While the Semantic Web is data-oriented and the World Wide Web is document-oriented, both are fundamentally decentralised, heterogeneous, and open: anyone can make any statement at any location, using any vocabulary or structure. In contrast, as shown in Table 1, traditional database-driven Web applications are typically centralised, with a fixed schema, a fixed vocabulary and a single data source.

Web applications	Semantic Web
centralised	decentralised
one fixed schema	semi-structured
one fixed vocabulary	arbitrary vocabulary
centralised publishing	publish anywhere
one datasource	many distributed datasources
closed world	open world

Table 1. Traditional and Semantic Web data

The conceptual and physical decentralisation of the Semantic Web can lead to (i) naming differences, since one person might describe the "author" of a book, a second the "writer", and a third the "creator"; to (ii) differences in data structures, since one person might describe his language skills in his personal profile, another person his pets, and a third his favourite colours; and to (iii) federated storage, since statements can simply be published to any Web location without central registration. In contrast to typical relational database applications, the

[1] `http://sioc-project.org`
[2] `http://w3.org/RDF/`
[3] `http://www.w3.org/TR/rdf-schema/`

Semantic Web has no central data repository, no central agreement on meaning, no central policy on terminology, and no necessary agreement on structure.

3.3 Extending existing Web application frameworks

Frameworks for building Web applications, such as Struts[4], Ruby on Rails[5] and Django[6] are a popular way to develop Web applications such as our example online communities application. These frameworks overcome the traditional problem of Web scripting languages, the intermixing of business logic, presentation templates markup and database operations, by using the model–view–controller (MVC) pattern (4).

The MVC pattern separates an application into three parts: the application *model* manages data representation and business logic, the *views* present the data and manage user interaction, and the *controller* handles control flow. The Web application frameworks provide, for each part of the MVC pattern, middleware libraries that support application development. Given the different nature of the Semantic Web, some of these libraries can be reused directly but additional provisions must also be made:

Models: from relational data to Semantic Web graphs Existing frameworks rely mostly on a direct object-relational mapping to automatically construct the models in the MVC pattern from the relational schema (5). But Semantic Web data does not follow one fixed schema and some data might not be described by any schema. For example, when integrating data from online communities we will encounter descriptions outside of the SIOC schema, such as concept hierarchies using SKOS[7], user profiles using FOAF[8], or project descriptions using DOAP[9]. The use of such additional information is explicitly advocated by SIOC, since SIOC itself only describes basic relations between online communities. SIOC exporters are encouraged to use additional terms to express further information about their users or their discussion topics.

Views: navigating large and arbitrary datasets In existing frameworks application developers construct the navigation interface manually, although aided by more abstract HTML template languages. Such navigation interfaces are almost always limited to the application's domain model and restricted to data that the developers initially anticipated. But on the Semantic Web we can encounter arbitrary data outside of our initially expected schema, so additional provisions are necessary to allow navigation based on that data. For example,

[4] `http://struts.apache.org`
[5] `http://rubyonrails.org`
[6] `http://www.djangoproject.com/`
[7] `http://www.w3.org/2004/02/skos/`
[8] `http://foaf-project.org/`
[9] `http://usefulinc.com/doap/`

we would like to browse our integrated community information chronologically (by time of posting) and by the author of the post. Such properties are part of SIOC and we can thus manually create a navigation interface for them. But if we encounter richer author profiles including the workplace of authors, their country of origin, or their field of expertise (none of which are in the SIOC schema) we would like to filter the posts based on these properties as well.

4 Developing the SIOC browser prototype

To develop an integrated solution for the online communities scenario described in Sect. 2, we extended the Ruby on Rails framework with components for consuming and processing Semantic Web data. One such component is ActiveRDF (1), which addresses the “model” mismatch and maps RDF data onto objects. The second component is BrowseRDF (6), a faceted browsing engine that enables navigation of large Semantic Web datasets without domain-specific navigation knowledge. The third component is a SIOC crawler which crawls, extracts, normalises, and integrates SIOC data from various community sites (which use different methods of exposing and linking to their SIOC data).

4.1 Augmenting Ruby on Rails

Ruby on Rails is an MVC-based rapid application development framework (7). Developers can generate models, views, and controllers matching their data, and can customise these to implement their business and domain logic. The model is typically provided by an automatic mapping from an existing database, the controller describes the control-flow in Ruby and the view is specified through HTML templates with embedded Ruby code.

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

Each of our extension components is designed to augment and integrate with Ruby on Rails. ActiveRDF can serve as a data layer in Ruby on Rails, replacing or augmenting the default ActiveRecord layer. The BrowseRDF navigation algorithms are implemented as a library that provides generic navigation on top of ActiveRDF; the library also includes helpers that generate the appropriate HTML navigation code in any Ruby on Rails application.

The SIOC crawler uses several libraries and command-line tools; cURL[10] and Hpricot[11] to extract links to the SIOC RDF data from the community sites; the

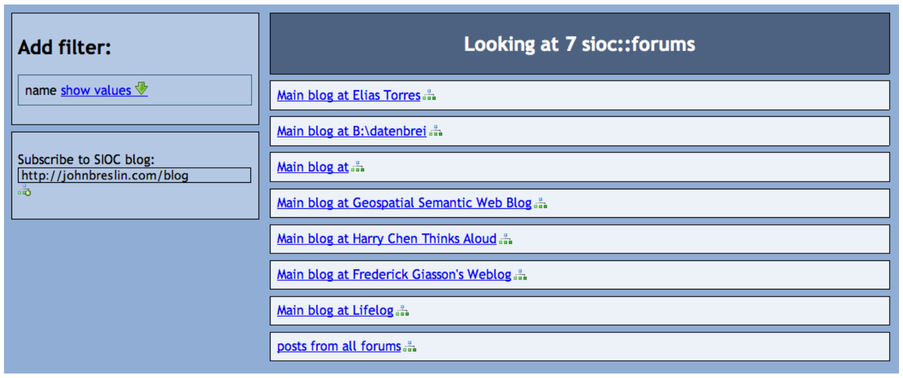
[10] `http://curl.haxx.se/`

[11] `http://code.whytheluckystiff.net/hpricot/`

Redland (8) “rapper” utility to fetch the actual SIOC RDF and to normalise it into ntriples; the Linux “cron” daemon to schedule periodic updates of the data; and Berkeley DB[12] as a persistent hashtable of visited sites.

4.2 SIOC explorer

Our prototype “SIOC explorer” aggregates data from various online community sites and allows users to browse and explore all disparate information in an integrated manner. The prototype, online at `http://activerdf.org/sioc`, source code at `http://launchpad.net/sioc-ex`, can be used as a feed reader to explore and subscribe to SIOC-enabled community sites such as weblogs, mailing lists, forums and IRC chats. The SIOC-enabled sites export SIOC data in a similar manner as RSS feeds. When prompted by the user, our application “subscribes” to such a feed and regularly polls these sites for updated content.


Add filter:

name show values

Subscribe to SIOC blog:

http://johnbreslin.com/blog

Looking at 7 sioc::forums

Main blog at Elias Torres

Main blog at B:\datenbrei

Main blog at

Main blog at Geospatial Semantic Web Blog

Main blog at Harry Chen Thinks Aloud

Main blog at Frederick Giasson's Weblog

Main blog at Lifelog

posts from all forums

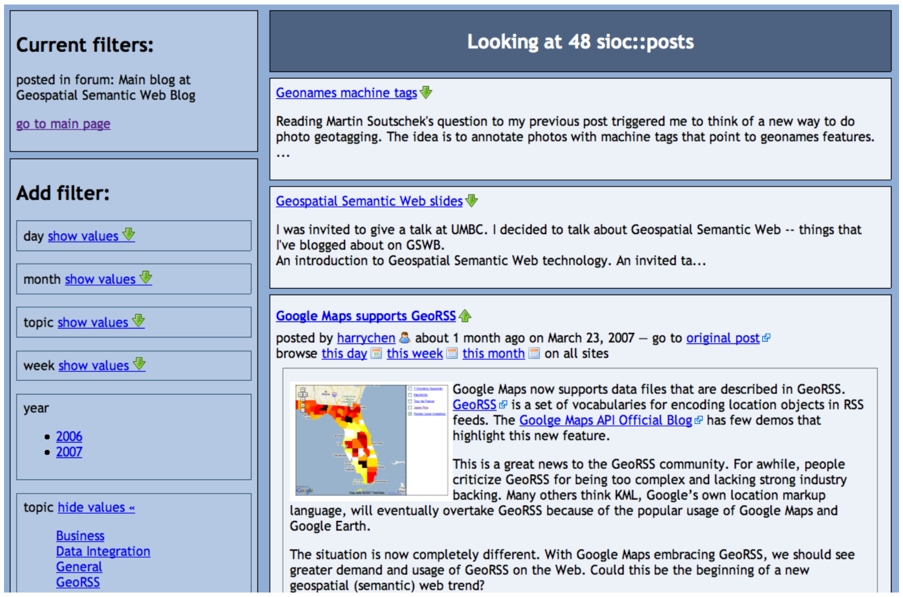
Fig. 1. Overview page of the SIOC explorer

All SIOC content is integrated into a local RDF store and then displayed in various ways. Figure 1 shows the overview page of the current prototype, with a list of several weblogs and forums. Users can decide to browse a particular forum, or see all posts aggregated from all sites. Not only posts from weblogs are shown, but also posts from from online community forums, IRC chats, mailing lists, etc. which are all described using the same SIOC RDF vocabulary.

After selecting a particular forum, the user is presented with the list of posts in that forum in the reverse chronological order, as shown in Fig. 2. As usual in feed readers, each post is summarised and can be expanded to read the full content. Also, “lateral” browsing is supported: clicking on the creator of a post shows all posts (including replies) written by this person, across all forums; clicking on a topic shows all posts tagged with this topic, again across all forums. In contrast to ordinary readers, our lateral browsing works across all types of

[12] `http://www.oracle.com/database/berkeley-db.html`

community forums: clicking on the user "Cloud" will not only show all his weblog posts, but also his emails from him and his contributions to IRC. The SIOC ontology enables this integrated browsing by providing the conceptual framework for unifying the content from the various community sites.


Current filters:

posted in forum: Main blog at Geospatial Semantic Web Blog

go to main page

Add filter:

day show values

month show values

topic show values

week show values

year

- 2006
- 2007

topic hide values «

Business
Data Integration
General
GeoRSS

Looking at 48 sioc::posts

Geonames machine tags

Reading Martin Soutschek's question to my previous post triggered me to think of a new way to do photo geotagging. The idea is to annotate photos with machine tags that point to geonames features. ...

Geospatial Semantic Web slides

I was invited to give a talk at UMBC. I decided to talk about Geospatial Semantic Web -- things that I've blogged about on GSWB.
An introduction to Geospatial Semantic Web technology. An invited ta...

Google Maps supports GeoRSS

posted by harrychen about 1 month ago on March 23, 2007 – go to original post
browse this day this week this month on all sites

Google Maps now supports data files that are described in GeoRSS. GeoRSS is a set of vocabularies for encoding location objects in RSS feeds. The Goolge Maps API Official Blog has few demos that highlight this new feature.

This is a great news to the GeoRSS community. For awhile, people criticize GeoRSS for being too complex and lacking strong industry backing. Many others think KML, Google's own location markup language, will eventually overtake GeoRSS because of the popular usage of Google Maps and Google Earth.

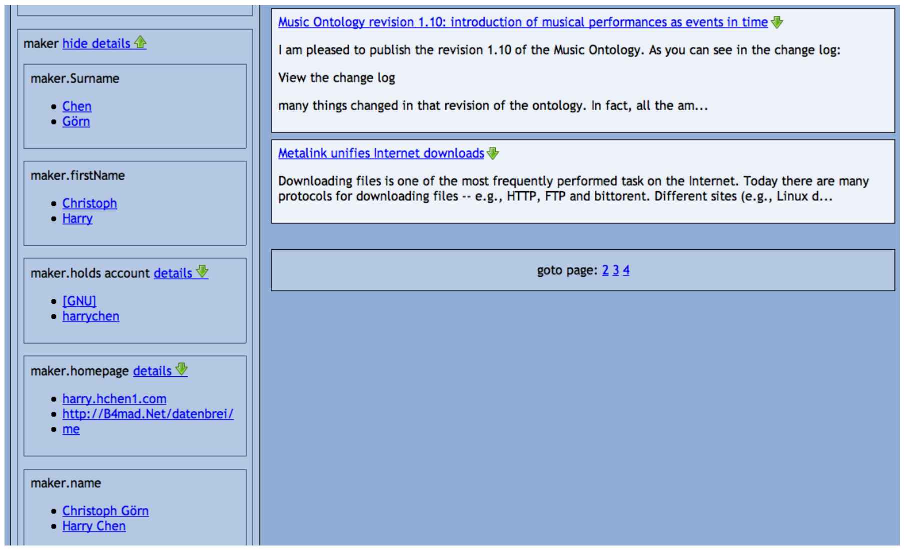
The situation is now completely different. With Google Maps embracing GeoRSS, we should see greater demand and usage of GeoRSS on the Web. Could this be the beginning of a new geospatial (semantic) web trend?

Fig. 2. Reading posts in the SIOC explorer

Finally, a generic faceted navigation interface is offered on the left-hand side, displaying relevant facets that are not part of the default interface. For example, since browsing posts by creation date, user, or topic is already supported through the "lateral" browsing discussed previously, those facets are not available on the navigation bar. But if the imported SIOC data (in this particular screenshot) has some unexpected facets, we can browse the posts based on e.g. creator, modification date, detailed user profiles (more than a simple username) and topic description (more than a simple topic tag).

Some facets (like the year) contain only "simple" values while others, such as maker or topic, can be further expanded to see subsequent subfacets. Fig. 3 shows the values for the subfacets of the facet "maker" for two different persons with posts about the topic "semantic web" in the database.

Application developers can customise the facet navigation to their needs and for example choose to exclude or include certain facets, or choose to exclude certain advanced operators such as the inverse join or the existential join.


maker hide details

maker.Surname
- Chen
- Görn

maker.firstName
- Christoph
- Harry

maker.holds account details
- [GNU]
- harrychen

maker.homepage details
- harry.hchen1.com
- http://B4mad.Net/datenbrei/
- me

maker.name
- Christoph Görn
- Harry Chen

Music Ontology revision 1.10: introduction of musical performances as events in time

I am pleased to publish the revision 1.10 of the Music Ontology. As you can see in the change log:

View the change log

many things changed in that revision of the ontology. In fact, all the am...

Metalink unifies Internet downloads

Downloading files is one of the most frequently performed task on the Internet. Today there are many protocols for downloading files -- e.g., HTTP, FTP and bittorent. Different sites (e.g., Linux d...

goto page: 2 3 4

Fig. 3. Faceted browsing in the SIOC explorer

4.3 Development effort

Using ActiveRDF and our other extensions, the integration of Rails with RDF data was straightforward and the development effort was quite low. The models itself are automatically provided as virtual models, the controller (with all application logic) contains around 95 lines of code and the views contain around 100 lines of abstract HTML. The SIOC crawler consists of around 150 lines of code. Most control-flow handling (such as routing HTTP requests) and interface code (such as Javascript generation) is provided by the Rails libraries.

5 Conclusion

Since online Semantic Web applications are based on existing Web infrastructure, developing these applications could leverage experiences with and infrastructure of existing frameworks. Using existing frameworks abstracts typical implementation patterns and shifts implementation effort from the developer to the framework.

As we have discussed, existing frameworks, many of which are based on the model–view–controller paradigm, cannot be reused completely for Semantic Web applications without resolving additional requirements. On the “model” part, the impedance mismatch between Semantic Web data and object-oriented programming needs to be resolved. On the “view” part, next to the domain-specific navigation, an automatic domain-independent navigation interface is needed that is not restricted to a specific schema.

We have introduced our extensions to the Ruby on Rails framework and discussed how they cater for these requirements. As a practical scenario, we have shown how this extended frameworks supports developing a SIOC browser. This SIOC browser integrates information from heterogeneous communities, allowing users to explore this information and find relevant posts and topics across these sites without the need to manually visit the different sites. The application consists of around 350 lines of (manually written) code, apart from the generated Rails code, which could be considered quite little for its functionality.

Acknowledgements This material is based upon works supported by the Science Foundation Ireland under Grants No. SFI/02/CE1/I131 and SFI/04/BR/CS0694.

References

[1] Oren, E., Delbru, R., Gerke, S., Haller, A., Decker, S.: ActiveRDF: Object-oriented semantic web programming. In: Proceedings of the International World-Wide Web Conference. (2007)
[2] Berners-Lee, T.: Weaving the Web. Collins (2000)
[3] Breslin, J., Harth, A., Bojars, U., Decker, S.: Towards semantically-interlinked online communities. In: Proceedings of the 2nd European Semantic Web Conference. (2005)
[4] Reenskaug, T.: Thing-Model-View-Editor, an example from a planning-system. Technical report, Xerox PARC (1979)
[5] Fowler, M.: Patterns of Enterprise Application Architecture. Addison-Wesley (2002)
[6] Oren, E., Delbru, R., Decker, S.: Extending faceted navigation for RDF data. In: Proceedings of the International Semantic Web Conference. (2006)
[7] Thomas, D., Hansson, D.H.: Agile Web Development with Rails. 2nd edn. Pragmatic Programmers (2007)
[8] Beckett, D.: The design and implementation of the Redland RDF application framework. Computer Networks **39**(5) (2002) 577–588

Extensible SPARQL Functions With Embedded Javascript

Gregory Todd Williams

University of Maryland, College Park, MD, USA
gtw@cs.umd.edu

Abstract. The SPARQL Query Language allows filtering of query results through arbitrary predicate expressions. Such expressions may invoke custom functions identified with IRIs, but the SPARQL implementation used must support the identified function. We present an extensible approach to allowing arbitrary function implementations using functions identified with URLs. We provide an example using Javascript functions, show how such a system can be implemented in the Perl based RDF::Query SPARQL implementation and discuss security concerns.

1 Introduction

The SPARQL query language[7] has provided a powerful, standardized way to query RDF models. Many query engine implementations exist that support SPARQL[4], and they allow consistent querying regardless of implementation language or platform. One of the most exciting features of SPARQL is its support for extension functions. These functions, identified by IRI, allow SPARQL queries to rely on domain-specific approaches to filtering query results.

Extension functions provide a means of extending the SPARQL language, but support for such functions remains sparse. Furthermore, to allow SPARQL query engines to share support for specific extension functions, implementations of the functions must be provided in each of the programming languages used.

We present a system in which extension functions are implemented in an agreed upon programming language and implementations may be shared among query engines by using an embedded interpreter for the language. Functions are identified by URLs (a subset of IRIs) and the source code may be retrieved at run time by dereferencing the URL. Such a system reduces duplicated effort in implementing extension functions, and allows reference implementations to define the semantics of functions.

2 Background

To make use of SPARQL extension functions, SPARQL query engines must implement functionality to allow users (or administrators) to register functions with the query engine, mapping an IRI to a function. Unfortunately, many query engines don't yet support this feature.

One popular query engine that does support this feature is ARQ[8] which allows querying of Jena[6] models. Functions may be registered with ARQ by installing a mapping between a URI and and a Java factory class. As a convenience shortcut to avoid the registering process, ARQ allows functions to be automatically loaded by using a URI with the "java:" IRI scheme. For example, using functions in the `java:com.hp.hpl.jena.query.function.library.` namespace will automatically make use of functions in the `com.hp.hpl.jena.query.function.library` package. Figure 1 shows an example SPARQL query with an extension function using the "java:" IRI scheme. Under ARQ, this example would dynamically load and execute the `com.ldodds.sparql.Distance` function, filtering any potential results whose corresponding return value was not less than a constant value.

```
PREFIX foaf:    <http://xmlns.com/foaf/0.1/>
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX geo:     <http://www.w3.org/2003/01/geo/wgs84_pos#>
PREFIX ldodds:  <java:com.ldodds.sparql.>
SELECT ?image ?placename
WHERE {
  ?image a foaf:Image .
  ?image dcterms:spatial ?point .
  ?point foaf:name ?placename .
  ?point geo:lat ?lat ; geo:long ?long .
  FILTER(
    ldodds:Distance(?lat, ?long, 38.9937, -76.933) < 10
  ) .
}
```

Fig. 1. An example ARQ SPARQL query finding all images taken within 10 kilometers of a known point using the `com.ldoddds.sparql.Distance` extension function.

Another query engine that supports extension functions is the Perl-based RDF::Query[10] package. In RDF::Query, IRIs may be mapped to user functions using the `add_function` object or class method to register the function either per query or globally for all queries, respectively.

Extension functions may be used in several different ways. Some extension functions are used as shorthand for accessing complex RDF structures. `jena:listMember` is used to test for membership in an `rdf:Seq` sequence[6]. Other extension functions are used to transform scalar values; `jena:sha1sum` is used to return the SHA-1 cryptographic hash of a value. A third type of function generates a new value from sets of values. `ldodds:Distance` can be used to compute the distance in kilometers between two sets of latitude/longitude values[5].

Despite having broad usefulness, these functions and many others are only available on one or a few query engine implementations, restricting their use.

In some situations, the need to use such a function may be addressed by using a specific query engine. However, in situations where a (potentially large) database is hidden behind a SPARQL query engine ”endpoint” (where an existing SPARQL engine is the only point of access to the underlying data), the user may be forced to use the query engine that is provided.

What is needed is an approach to extension functions that allows functions to be implemented once, and shared between query engines without *a priori* knowledge of any particular function.

3 Methodology

Our system extends the RDF::Query query engine with the ability to retrieve Javascript extension function implementations via URL, run the extension functions in an embedded Javascript interpreter, and return the resulting value to the RDF::Query engine as a native Perl object. Javascript was chosen for its wide use on the internet, existing RDF APIs for Javascript, and the availability of several high quality, free implementations suitable for embedding [1, 9]. However, our approach is general, and could work with any embeddable language.

Our system makes use of an RDF schema to describe SPARQL extensions, a SPARQL extension function API for Javascript, and optional cryptographic signatures to distinguish ”trusted” implementations. We discuss these components below, together with notes on the actual implementation in RDF::Query.

3.1 Extension Function Schema

The extension function namespace is:

```
http://www.mindswap.org/2007/owl/sparql#
```

The RDF document describing functions is used as a central location for information about the implementation of the functions. It should contain data relating to the location of implementation files and any cryptographic signatures, a description of the referenced functions, a name for each function (to be used as an entry point into the code and that will return the function value), and any other information relevant to the implementation.

An example RDF document describing two of the jena functions using the extension function schema is shown below in the Turtle[2] syntax:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix ex: <http://www.mindswap.org/2007/owl/sparql#> .
@prefix dc: <http://purl.org/dc/elements/1.1/> .

<java:com.hp.hpl.jena.query.function.library.>
    a ex:Namespace;
    ex:hasFunction
        <java:com.hp.hpl.jena.query.function.library.sha1sum>,
        <java:com.hp.hpl.jena.query.function.library.now> .

<java:com.hp.hpl.jena.query.function.library.sha1sum>
    a ex:Function;
    dc:description "SHA-1 Hash";
    ex:source <http://example.com/sha1.js>;
    ex:signature <http://example.com/sha1.js.asc>;
    ex:function "sha1" .

<java:com.hp.hpl.jena.query.function.library.now>
    a ex:Function;
    dc:description "Start time of query execution";
    ex:source <http://example.com/now.js>;
    ex:signature <http://example.com/now.js.asc>;
    ex:function "now" .
```

The important statements of this RDF description for purposes of implementation are those using the following predicates:

- `ex:source` declares the location of the implementation source code.
- `ex:function` defines the function name in the source code that should be called to execute the function.
- Optionally, `ex:signature` declares the location of any cryptographic signatures that can be used by query engines in order to run only trusted functions. This is discussed in more detail in sections 3.3 and 4.1.

3.2 Extension Function API

The implementation of extension functions must take as input RDF nodes passed as function arguments and output an RDF node value. Therefore, a simple API for interacting with RDF nodes is needed in the implementation language. We use the AJAR[3] Javascript API, providing the classes `RDFLiteral`, `RDFBlankNode`, and `RDFSymbol`, the common property `termType`, the common method `toString`, and the node-creating function `makeTerm`. For a more detailed description of the API, refer to the AJAR documentation.

Figures 2 and 3 show corresponding RDF description and Javascript implementation of a distance function using the Javascript API. This function is similar to `ldodds:Distance`, taking two latitude/longitude pairs, and returning the distance between them in kilometers as computed using the great circle distance algorithm.

3.3 Cryptographic Signatures

For security and performance reasons, it may be desirable for query engine instances to only allow running extension functions that have been identified as trusted. This can be accomplished by designating known third-parties as trusted,

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix ex: <http://www.mindswap.org/2007/owl/sparql#> .
@prefix dc: <http://purl.org/dc/elements/1.1/> .

<http://example.com/functions/>
    a ex:Namespace;
    ex:hasFunction <http://example.com/functions/distance> .

<http://example.com/functions/distance>
    a ex:Function;
    dc:description "Geographic (great circle) distance in kilometers";
    ex:source <http://example.com/distance.js>;
    ex:function "gcdistance" .
```

Fig. 2. RDF description of a distance extension function.

```
function gcdistance( lat1, lon1, lat2, lon2 ) {
  lat1  = deg2rad( makeTerm(lat1).toString() );
  lat2  = deg2rad( makeTerm(lat2).toString() );
  lon1  = deg2rad( makeTerm(lon1).toString() );
  lon2  = deg2rad( makeTerm(lon2).toString() );

  var londiff = Math.abs(lon1 - lon2);
  var s1    = square(Math.sin((lat2 - lat1) / 2));
  var s2    = square(Math.sin( londiff / 2 ));

  var sq  = Math.sqrt(
            s1
            + Math.cos(lat1)
            * Math.cos(lat2)
            * s2
          );

  var adist = 2 * Math.asin( sq );
  var r     = 6372.795;
  return r * adist;
}

function square (x) { return x * x; }
function deg2rad(d) { return Math.PI*d/180 }
```

Fig. 3. Javascript implementation of a distance extension function.

and trusting any implementation that has a valid cryptographic signature from the trusted third-party.

If configured to allow only trusted functions, our implementation takes a list of trusted GPG public key fingerprints, attempts to verify all available signatures of a requested implementation, and only proceeds if there exists a signature created with a trusted key.

The need for a possibly growing list of arbitrary signatures in the RDF description is the primary reason why function metadata and implementation source code must reside at distinct URLs. As new signatures are generated, metadata describing the signatures must be added to the RDF description. However, this addition must not invalidate existing signatures. Therefore, signatures must be made against the implementation source code and referenced in the seperate function metadata RDF document.

3.4 Extending RDF::Query

We implemented this system in the Perl-based RDF::Query query engine using an embedded Javascript interpreter. During query execution, an extension function is retrieved as a code reference via a lookup method. This method first looks for a native implementation of the extension function. If no implementation is found, and the function is identified with a URL, the URL is dereferenced and the resulting content is interpreted as an RDF document describing the function implementation. Using this RDF data, the implementation source code is retrieved, compiled, and verified against any signature files. If this process is successful, a code reference is returned that will behave just as a native implementation would.

Figure 4 shows example code for constructing an RDF::Query query engine, enabling the use of URL-based extension functions, and restricting their use to only those signed by a known key. In this example, a Javascript implementation of the SHA-1 hashing function is used to find all people with an email address that hashes to a known value.

In the construction of the query engine object, two new constructor fields are introduced to support the URL-based extension functions:

– `net_filters` is passed with a true value to enable URL-based extension functions.
– `trusted_keys` is passed with an array reference containing a list of the trusted signing keys' cryptographic fingerprints. This field is optional.

After the first use of an extension function, the source code and signatures are cached to prevent unnecessary repeated downloads. Using features of HTTP/1.1, the content of the source and signatures need only be downloaded again if these documents change. On repeated requests where the documents have not changed, a response with only a small header is transfered. This approach saves bandwidth while ensuring that changes to the source code and signatures are not ignored due to arbitrarily long caching – an important feature for long running query

```
my $sparql = <<"END";
  PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
  PREFIX foaf: <http://xmlns.com/foaf/0.1/>
  PREFIX func: <http://example.com/functions.rdf#>
  SELECT ?person
  WHERE {
    ?person a foaf:Person ;
    foaf:mbox ?mbox .
    FILTER(
      func:sha1(?mbox) = "f80a0f19d2a0897b89f48647b2fb5ca1f0bc1cb8"
    ) .
  }
END
my $query = RDF::Query->new( $sparql, undef, undef, 'sparql',
  net_filters  => 1,
  trusted_keys => ['1150 BE14 FF91 269F 398B 0F4E 0253 5AF9 A2B9 659F'],
);
```

Fig. 4. Setting up an RDF::Query query engine with support for network-based extension functions only when signed by a known key.

engines. Furthermore, if the source code was verified against a signature, the verified status of the code may be cached when using the same trusted key and source code to avoid unnecessary repeat verifications.

4 Results and Analysis

The implementation of URL-based extension functions in RDF::Query was relatively straightforward. Just sixteen lines of existing code were modified, and three new methods were added to the main query engine class to initialize an embedded Javascript interpreter and to compile and call the extension function. There are several issues of concern with such an implementation relating to potential security and performance degredation. These issues are discussed below.

4.1 Security and Performance Concerns

With many SPARQL endpoints being made publicly available for use by the public, thought must be given to the potential to abuse these systems. Relating to URL-based extension functions, such abuse could affect overall system performance, or even cause a denial of service attack against external machines.

Query engines that may run unknown code loaded as an extension function must be aware of the potential problems doing so may cause. A denial of service attack may be initiated against a foreign server by using an extension function URL pointing to a very large resource (in which case available local bandwidth may be a concern) or a resource that is computationally intensive to return (where the total system load of the remote server may be a concern). Similar attacks may be initiated if the embedded Javascript environment allows the creation of network connections.

Another potential problem of running arbitrary code is local system load. If an extension function runs for an unexpectedly long time, it may prevent the server from answering other requests or may overrun a time limit on the request (as might be likely if the request was made using HTTP). Even if the running time of a function is bounded, system load may still be of concern if the overhead of switching contexts between the query engine and the embedded language is high. If the potential result set before filtering is large, and many calls to the extension function are necessary, the context switching overhead may overwhelm the total running time of the query.

Public endpoints may prevent many potential problems by allowing only extension functions signed by trusted keys. By relying on a trusted third party to sign a function, the signature may be used to indicate the correctness of the function (by a lack of syntactic and semantic errors) as well as the absence of malicious code. This approach may work well for domain-specific functions where a central organization or interest group could provide the trusted signatures.

To address denial of service attempts and performance concerns, a query engine may also define configurable maximums for source code size and run time. In situations where a maximum run time isn't acceptable (where query results must be provided despite run time), total run time and the overhead of context switching may be addressed by providing a native implementation of commonly used extension functions. Native implementations may also be important for functions where a native implementation can provide much better performance (due to more efficient data structures, access to hardware acceleration, etc.).

5 Conclusion and Future Work

We believe this approach to allowing arbitrary extension functions can provide commonly used functions to many query engines with comparatively little work. However, to realize this potential and to be generally useful, our approach would need implementation in more than just one query engine.

Future work is needed to implement parts of the Javascript API to allow extension functions to access the underlying RDF model, allowing implementation of model-dependant functions such as jena:listMember. Future work may also include extending the RDF schema to allow for alternate programming language and cryptographic signature types. A language designed specifically for embedding (such as Lua) may prove more desirable for the implementation of functions, while allowing other signature types (x.509 certificates, for example) could allow leveraging existing key infrastructures. Finally, we hope to follow this work with a deeper investigating of the real-world applications and benefits of such a system.

This work was supported by grants from Fujitsu, Lockheed Martin, NTT Corp., Kevric Corp., SAIC, the National Science Foundation, the National Geospatial - Intelligence Agency, DARPA, US Army Research Laboratory, and NIST.

References

1. Webkit. http://developer.apple.com/opensource/internet/webkit.html.
2. Dave Beckett. Turtle - Terse RDF Triple Language. http://www.dajobe.org/2004/01/turtle/.
3. Tim Berners-Lee, Yuhsin Chen, Lydia Chilton, Dan Connolly, Ruth Dhanaraj, James Hollenbach, Adam Lerer, and David Sheets. Tabulator: Exploring and analyzing linked data on the semantic web. In *3rd International Semantic Web User Interaction Workshop*, November 2006.
4. Christian Bizer and Daniel Westphal. Update on semantic web toolkits for scripting languages. In *2nd Workshop on Scripting for the Semantic Web (SFSW2006)*, June 2006.
5. Leigh Dodds. Sparql geo extensions. http://xmlarmyknife.org/blog/archives/000281.html.
6. Brian McBride. An introduction to rdf and the jena rdf api. http://jena.sourceforge.net/tutorial/RDF_API/.
7. Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF (Working Draft). http://www.w3.org/TR/rdf-sparql-query/.
8. Andy Seaborne. ARQ - A SPARQL Processor for Jena. http://jena.hpl.hp.com/ afs/ARQ/.
9. Jens Thiele. Embedding spidermonkey - best practice. http://egachine.berlios.de/embedding-sm-best-practice/embedding-sm-best-practice.pdf.
10. Gregory Todd Williams. RDF::Query. http://search.cpan.org/dist/RDF-Query/.

A lookup index for Semantic Web resources

Eyal Oren and Giovanni Tummarello

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

Abstract. Developers of Semantic Web applications face a challenge with respect to the decentralised publication model: where to find statements about encountered resources. The "linked data" approach, which mandates that resource URIs should be de-referenced and yield metadata about the resource, helps but is only a partial solution and not followed widely. We present a simple lookup index that crawls and indexes resources on the Semantic Web. Our index allows applications to automatically retrieve sources with information about a certain resource. In contrast to more feature-rich Semantic Web search engines, our index is limited in scope and functionality and is therefore simple, small, and scalable.

1 Introduction

The Semantic Web can be seen as a large knowledge-base of statements about resources. These statements form an interconnected graph since statements may mention the same resources as other statements. A fundamental feature of the Semantic Web, as opposed to older forms of semantic networks, is that the graphs are decentralised: there is not one single knowledge-base that contains the graph of statements but instead anyone can contribute statements on his "personal" web-space. The complete graph is only visible after crawling and integrating the fragments mentioned on these personal subspaces.

This decentralised nature of the Semantic Web is well-known and, similarly to the decentralised nature of the ordinary Web, actually one of its benefits: anyone can make any statement about anything without the need for centralised control or authorisation. But for developers of Semantic Web applications, which operate on Semantic Web data, the decentralisation poses a challenge: *how and where to find statements about certain resources*?

This paper introduces Sindice, a simple lookup index that helps application developers answer exactly this question. The index crawls the Semantic Web and indexes the resources encountered in each source. A simple HTTP API returns sources containing statements about a given resource, ranked in order of relevance. In contrast to full-blown "Semantic Web search engines" we have a purposely limited scope: we only index RDF documents and we only index occurrences of resources; as a result, our code is simple and our index is relatively small.

2 Sindice architecture

Our Sindice index maintains a list of resources and the sources in which these resources appear (either as subject or as object of a statement). The basic functionality of the index is to return such a list of sources for any given URI, so as to allow the requesting application to visit these sources and fetch more information about the resource in question.

The service is provided for usage in Semantic Web applications, to implement for example a "find more information" button which would find additional sources of information through Sindice. Our lookup service augments the "linked data model" which mandates that information about an RDF resource should be available on the URI identifying the resource. Sindice promotes this approach (in its ranking algorithm) but also supports URIs that are not URLs and cannot be dereferenced (such as telephone numbers, isbn numbers or general concepts); most importantly, Sindice helps locating statements made outside the "authoritive" source for a resource.

2.1 Design requirements

The architecture for our Sindice index consists of three components for indexing, lookup, and ranking of resources. These three components provide the three methods in the Sindice API:

- `lookup(uri) => url[]`: lookup of a URI returns a ranked list of URLs in which the given URI has been mentioned,
- `index(url) => nil`: parses and indexes document at given URL,
- `refresh() => nil`: refreshes the index by re-visiting and updating all known sources.

We have two design requirements: first, we want to minimise the index size, so as to allow indexing of the whole Semantic Web without requiring complex disk solutions such as networked storage, disk-arrays or clustered machines; secondly, we want to minimise lookup times, so as to allow applications to use Sindice by default to lookup more information for any encountered resource.

To fulfil these requirements, we focus on simplicity: we only index occurrences of resources but not the actual statements in which they occur; we only allow lookups from resources to sources but not in the other direction (i.e. one cannot ask which resources are mentioned by a particular source); we use a simple ranking algorithm that produce useful results without requiring much computation time or background knowledge.

2.2 Index design

We aim for storing at least 300 million resources, mentioned on maybe 3 million sources. These numbers serve only as an indication of our scalability target and are informed by earlier crawling results from SWSE[1] and Swoogle[2] @@ding/finin "characterising semantic web on the web". In designing the index, we optimise for disk space and lookup times. Since the only required access pattern is from resource to mentioning sources, a hashtable-like index seems natural. The key to such a hashtable would be the URI of the resource and the value of the hashtable would be the list (array) of mentioning sources.

In Listing 1.1 we describe the basic indexing algorithm, while abstracting for a moment from the exact implementation of the hashtable-like occurrence index. As shown in the listing, when indexing a new source, we extract each mentioned URI in that source and add it to the "occurrence" index to indicate that the indexed source mentions the found URI.

Listing 1.1. Indexing algorithm pseudo-code

```
def index(source)
  resources = subject_and_objects_in(source)
  for resource in resources
    occurrence[resource] += source
  end
end
```

With such a hashtable-like index, lookups are implemented by simple lookups in the hashtable and ranking of the results, as shown in Listing 1.2. Since the hashtable-like index uses the resource URI as key and the mentioning sources as values, average lookup complexity is $O(1)$. Worse-case lookups of frequently mentioned resources, especially frequent classes such as `foaf:Person` or `sioc:Forum`, would cause a higher runtime simply to read and return the large set of mentioning sources, but we do not envision these to be requested often.

Listing 1.2. Lookup algorithm pseudo-code

```
def lookup(resource)
  sources = occurrence[resource]
  return rank(sources)
end
```

2.3 Index implementation

In abstract sense our index design is clear, but we have several choices for concrete implementation of such a hashtable-like index. We have experimented with the following options:

Database: using a database table with the resource URIs as primary keys. The advantage is simplicity of implementation, the disadvantages are: that databases typically incur a slight overhead in query processing which we would not use since we have only simple key lookups, that many databases have upper limits on the amounts of rows which we could possibly reach, and most importantly, that databases typically need to have the complete

[1] `http://swse.deri.org`
[2] `http://swoogle.ubmc.edu`

list of primary keys in memory to ensure fast access to the data; in our case 300 million primary keys (resources that are each maybe 128 bytes to 500 bytes long) would imply main memory requirements of 38GB-150GB which is rather more than our standard available hardware offers.

Filesystem: implementing our own hashtable using the filesystem with one file for each resource containing the list of mentioning sources. The advantage of this solution is scalability: storing only a list of sources and compressing the contents of each file, each resource-file would occupy between 50 bytes to 300 bytes, resulting in an index size (disk-space) of 30GB which is well inside the range of current typical hardware. The disadvantage however, is that filesystems typically have a minimal block-size (configured at formatting time) which is an lower limit on occupied disk-space for each file. On for example the ext3 filesystem, typical block-size is 2K, so even if our files are only 50 bytes long, they would each occupy 2KB in diskspace; in the future we will experiment with filesystems without such limitations, such as Reiser4[3].

Persistent hashtable: using an existing library for persistent hashtables such as Berkeley DB. This solution is very similar to the second (persisting the hash-table directly onto the filesystem) except that Berkeley DB allows us to configure various parameters of the hash-table (bucket size, overflow rate, hash-function, etc.) and that Berkeley DB manages all the necessary internals to allow transaction management, concurrent access, locking etc.

After experimenting with these options and considering their characteristics, we implemented Sindice using the Berkeley DB persistent hashtable. We are still in the process of tuning it for best concurrent performance and disk space usage (e.g. bucket size and fill factor).

2.4 Source metadata

Additionally to the resource occurrence hashtable, we maintain a small hashtable with metadata for each visited source. This hashtable is used for managing the crawling and fetching process, and for the ranking (explained in the next section). To prevent over-requesting metadata sources and to follow the Robot (crawling) guidelines[4], we store visiting times and content hash for each source. We only revisit sources after a threshold waiting time and we only reparse the source's response if the content hash has changed. We do not yet use E-Tags and HTTP "last-modified" headers but consider doing so in the future.

2.5 Ranking phase

Since for popular resources our index could easily return many hundreds or thousands sources, a good ranking algorithm is crucial. But on the other hand,

[3] `http://www.namesys.com/v4/v4.html`
[4] `http://www.robotstxt.org/wc/guidelines.html`

we want to keep the index small, and thus amount of metadata minimal, we want to rank resources fast.

We have designed a ranking function that requires only little metadata for each source and is relatively fast to compute; in contrast to more optimal ranking functions such as HITS (Kleinberg, 1999), PageRank (Brin and Page, 1998), or ReconRank (Hogan *et al.*, 2006), it does not construct a global ranking of all sources but ranks sources based on their own metadata and external ranking services. We currently use the following metadata in ranking:

Hostname: we prefer sources whose hostname is the same as the resource's hostname. For example, we consider that more information on the resource `http://eyaloren.org/foaf.rdf#me` can be found at the source `http://eyaloren.org/foaf.rdf` than at an arbitrary "outside" source, such as `http://g1o.net/g1ofoaf.rdf`. We thus reuse the Internet's own authority mechanism and favour the notion of linked Semantic Web data, as advocated by e.g. Sauermann *et al.* (2007), in which resources use URIs that are resolvable and return valuable information.

Relevant statements we prefer sources that mention this resource more often than sources that mention this resource less times (considering that both sources mention the resource at all).

Source size: we prefer sources with more information than sources with little information.

External rank: we prefer sources hosted on sites with a high ranking in existing ranking services (such as Google's PageRank). We thus use techniques @@cite to estimate a ranking for each source at index time and assume that highly-ranked websites also produce valuable RDF data. For example, using an external rank the source `http://www.deri.ie/fileadmin/scripts/foaf.php?id=95` would rank higher than the source `http://g1o.net/g1ofoaf.rdf` for the same example resource `http://eyaloren.org/foaf.rdf#me`, because `http://deri.ie` has a higher estimated PageRank than `http://g1o.net`.

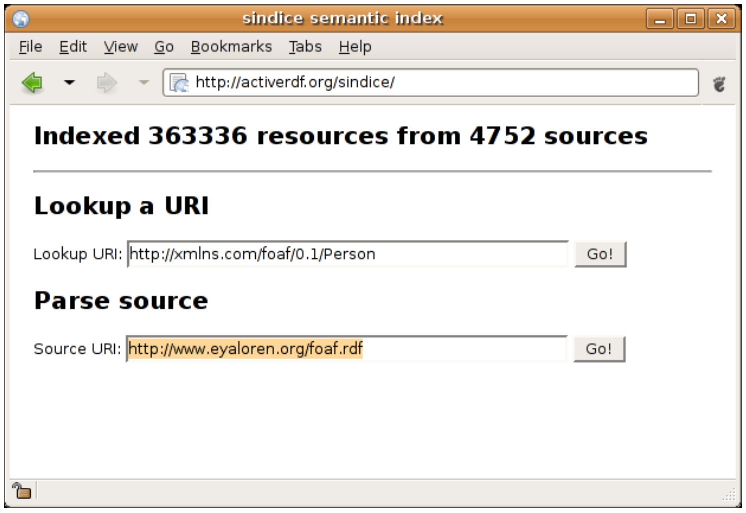
3 Current implementation

The current implementation is online at `http://sindice.com/`, the source code is available[5] under LGPL license.

To fetch and transform all RDF to canonical ntriples format, we use the Redland (Beckett, 2002) "rapper" utility: although we do not need to parse the RDF but only extract the resources mentioned, such extractions are difficult in XML since URIs can be mentioned without being true resources (e.g. as namespace declarations). For this reason, we employ rapper to parse all RDF into ntriples, and use a simple regular expression to extract all resource subjects and objects.

[5] `https://launchpad.net/sindice/`

To update our index, we use the pinging service `http://pingthesemanticweb.com`: we ask it periodically (currently every hour) for recently updated RDF documents and index each of these sources if they have changed since the last time we saw them. We also offer a Web interface and an HTTP API where people can submit their own RDF documents for indexing, but we prefer to reuse the `http://pingthesemanticweb.com` service since it already receives many pings from updated RDF documents.


sindice semantic index
File Edit View Go Bookmarks Tabs Help
http://activerdf.org/sindice/

Indexed 363336 resources from 4752 sources

Lookup a URI

Lookup URI: http://xmlns.com/foaf/0.1/Person Go!

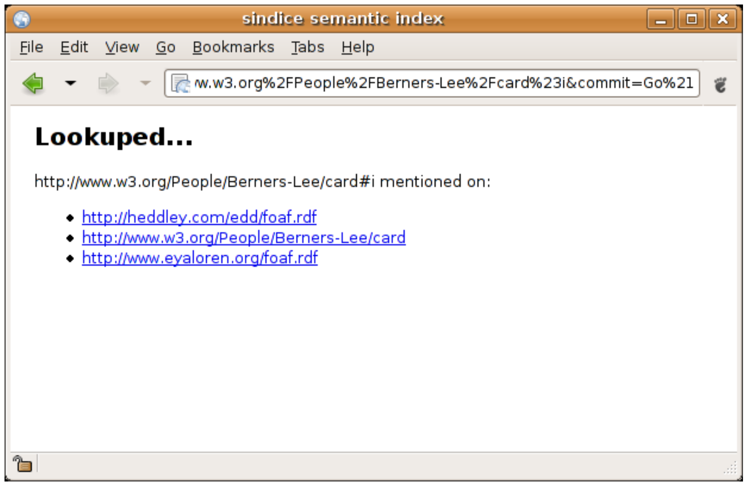
Parse source

Source URI: http://www.eyaloren.org/foaf.rdf Go!

Fig. 1. Overview page of Sindice

Figure 1 shows the human-readable Web interface for Sindice. It shows some index statistics and access to the two API functions: lookup of a resource and parsing or updated source. The third API function, refreshing all sources, is not accessible over the Web interface to prevent denial-of-service attacks. Apart from the Web interface Sindice offers an HTTP API interface that returns XML, JSON, or plain text results, through which developers can lookup sources from within their programs.

Figure 2 shows the resulting lookup of a sample resource, in this case `http://www.w3.org/People/Berners-Lee/card#i`. As can be seen in the shown results, ranking was not yet taken into account at the time of the screenshot. Lookups on the current prototype take around @@@0.025s in average (variance probably due to disk cache, network lag, processor load and the amount of sources returned).


sindice semantic index
File Edit View Go Bookmarks Tabs Help
w.w3.org%2FPeople%2FBerners-Lee%2Fcard%23i&commit=Go%21

Lookuped...

http://www.w3.org/People/Berners-Lee/card#i mentioned on:

- http://heddley.com/edd/foaf.rdf
- http://www.w3.org/People/Berners-Lee/card
- http://www.eyaloren.org/foaf.rdf

Fig. 2. Lookup of example resource

4 Related Work

We are aware of two Semantic Web search engines that index the Semantic Web by crawling RDF documents and then offer a search interface over these documents.

SWSE[6] crawls not only RDF documents but also "normal" HTML Web documents and RSS feeds and converts these to RDF (Harth *et al.*, 2007; Hogan *et al.*, 2007). SWSE stores the complete RDF found in the crawling phase and offers rich queries (expressiveness comparable to SPARQL) over this RDF data. Since SWSE also stores the provenance of all statements, it can also provide the source lookup functionality that we provide but with a cost: lookups are slower than in Sindice and the index is larger.

Similar to SWSE, Swoogle (Finin *et al.*, 2005) crawls and indexes the Semantic Web data found online. Again, the same differences apply: Swoogle offers richer functionality than we do but at a cost of index size and lookup times.

Finally, we compare our index to `http://pingthesemanticweb.com`. They maintain a list of recently updated documents, and are currently indexing over seven million RDF documents, but in contrast to our service, they do not index the statements or resources mentioned in these sources. Still, the service is very useful as companion to ours and indeed we use it to find recently updated RDF sources.

5 Conclusion

We have presented a simple lookup index for Semantic Web resources. Our index is small and scalable and allows for fast lookups. In contrast to other approaches,

[6] `http://swse.deri.org/`

our scope is purposely limited to keep the index simple and small. Future work and discussion in the workshop will be focused on improving performance and concurrency and evaluating and discussing the ranking approach.

Acknowledgements This material is based upon works supported by the Science Foundation Ireland under Grants No. SFI/02/CE1/I131 and SFI/04/BR/CS0694.

References

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

S. Brin and L. Page. Anatomy of a large-scale hypertextual web search engine. In *Proceedings of the International World-Wide Web Conference*. 1998.

T. W. Finin, L. Ding, R. Pan, A. Joshi, *et al.* Swoogle: Searching for knowledge on the semantic web. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*. 2005.

A. Harth, J. Umbrich, and S. Decker. Multicrawler: A pipelined architecture for crawling and indexing semantic web data. In *Proceedings of the International Semantic Web Conference (ISWC)*. 2007.

A. Hogan, A. Harth, and S. Decker. Reconrank: A scalable ranking method for semantic web data with context. In *Second International Workshop on Scalable Semantic Web Knowledge Base Systems*. 2006.

A. Hogan, A. Harth, J. Umbrich, and S. Decker. Towards a scalable search and query engine for the web. In *Proceedings of the International World-Wide Web Conference*. 2007. Poster presentation.

Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46, 1999.

L. Sauermann, R. Cyganiak, and M. Völkel. Cool URIs for the semantic web. Tech. Rep. TM-07-01, DFKI, 2007.

JAWS: A Javascript API for the Efficient Testing and Integration of Semantic Web Services

David A. Ostrowski

System Analytics and Environmental Science
Research and Advanced Engineering
Ford Motor Company
dostrows@ford.com

Abstract. Semantic Web Services (SWS) hold a lot of potential to the future of the Semantic Web. In this area, a number of tools have been developed to facilitate their definition and deployment. Our goal is to support an efficient means of testing and integration within a browser-based solution. For this purpose we propose JAWS (Javascript, AJAX, Web Service) : A Javascript API to facilitate the testing and integration of SWS. This software decouples the process of SWS integration and development through facilitation of the AJAX/REST paradigm. By leveraging meta-programming and deep integration techniques we support Web 2.0 inspired applications in the context of complete browser-based development.

Keywords: Semantic Web Service, Javascript, AJAX, REST

1 Introduction

General applications of web services support enterprise development as a means to reduce costs and complexity of integration.[1] This is accomplished through machine-independent protocols and utilization of internet-based technologies for communication. These advantages and characteristics provide a strong motivation for integrating this technology with the Semantic Web, most commonly regarded as the next generation of the Web.[2] This work intends to support this goal by leveraging existing toolkits through an API to enable the automatic and semi-automatic utilization of SWS. [3][4][5] Specifically, we are interested in the rapid facilitation of activities linked to SWS including matchmaking, input/output types comparison and analysis of effects. [6][7] Through the combination of Javascript, AJAX and Web Services we intend to satisfy a number of goals:

- **Web 2.0** We are interested in providing a higher level of accessibility to web services – supporting work collaboration and data sharing among second generation web applications.
- **Integration** Given the ubiquitous nature of Javascript, developers can readily incorporate a Javascript API within any web-based frameworks.
- **Deployment (Compatibility)** All software functionality is developed without browser-specific capability**.**
- **Protection** Methods of access to corporate (secure) data stores are controlled through the application of complete Javascript APIs. This approach to controlled data sharing has been popularized by such companies as Amazon and Google .[8][9]

- **Simplicity** Applications can be developed solely in Javascript requiring knowledge of only one language. Complexity of software integration (data server, external software) can be handled separately from a client side developer.

In section Two, we present an overview of our SWS environment including an introduction to the API functionality. Section Three discusses the major constructs of our API. Chapter Four demonstrates its usage in a short example along with a use-case scenario. Chapter Five concludes with a summary including several issues highlighted for future expansion within our API.

2 SWS Environment

The design for our SWS environment relies on two separate knowledge bases.(Figure 1) The first OWL-based KB supports the description of the application domain, defining concepts and terms used for web services description. The second maintains semantic based-definition of web services through the employment of OWL-S [10][11].
Our API supports interaction with these two data sources as well as invocation of the established web services. The first activity supports development of AJAX-based requests to support discovery of SWS within an OWL-based taxonomy. This task ranges from simple keyword matching within a class taxonomy to interaction with server-side tools to provide advanced query capabilities and reasoning. [12][13][14][15] Here, the JAWS API relies upon REST-based services providing efficient support to clients. The second major support step is to utilize AJAX-based requests to identify and retrieve OWL-S files in order to dynamically generate Javascript objects for their representation. By referencing the predefined format of the OWL-S files, users are able to generate applications for the purpose of comparison, integration and testing. The last step is the utilization of the Javascript constructs to invoke our REST-based web services. Here, the services will be defined in an array of Javascript objects referenced by the web services name. Javascript methods as named in OWL-S representations will be used to reference the REST-defined web services.

2.1 Software Layers

The theme of the JAWS architecture is to provide a browser-based environment that separates the activity of SWS testing and maintenance. With this goal the top layer is presented as the application layer in which a complete SWS application is developed either completely in Javascript or in cooperation with another framework. In this layer the user interface is HTML (also supporting 3D markup via embedded object such as in our case study) controlled via Javascript. In the second layer, we have the actual API that exists as a Javascript library. The third layer is considered the mapping layer in which specific OWL and OWL-S data are mapped to Javascript objects and methods on-the-fly. This development relies heavily on the leveraging of XHR alone to reference data stores residing as OWL and OWL-S files as well as integrating with

REST-based web services. OWL-defined data stores are read by adapters as REST web services enabling the client to perform activities related to SWS discovery. Due to minimal size, OWL-S documents are completely referenced by XHR requests. Finally, the web service invocation is presented as Javascript objects and methods allowing the client programmer to access the data. Web service implementations not conforming to the REST approach or outside of the current domain are handled by a process designed to make secondary web service calls to bridge them to our SWS environment.

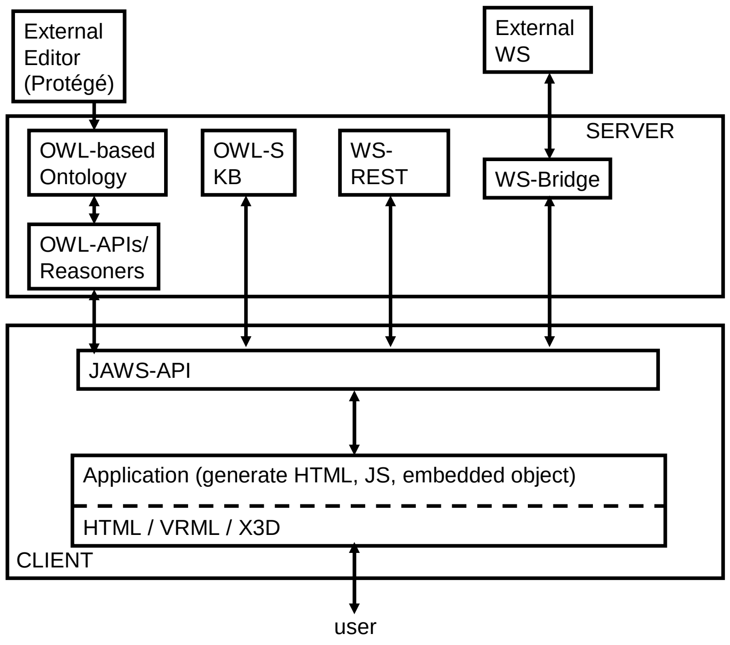


Figure 1 SWS Environment

2.2 Advantages of Javascript

Scripting languages (Ruby, Python, PHP) have gained traction in their application to map RDF-based resources to more programmer-friendly representations. [16][17][18] While reflection is supported in Java and attributes related to dynamic prototyping (interfaces) are supported in languages such as C++, they do not allow for the same level of flexibility in implementation. Maintaining the dynamic run-time capabilities of scripting languages, Javascript maintains similar advantages to Python or Ruby. Recently, with the utilization of the XMLHTTPRequest (XHR) object library, Javascript has demonstrated increased potential by matching capabilities held by traditional server-side scripting languages.[19][20] In utilization of the XHR request model, Javascript can supply a unique approach to the development of semantic web applications due to its support of asynchronous activity. While any web service implementation can fit into our architecture it is the REST design philosophy that demonstrates highest efficiency. Through the employment of the basic HTTP constructs, the REST approach supports a low-level means of web service implementation. Through avoidance of higher level constructs, REST calls are easily employed within XHR requests thus avoiding extra software for browser-based invocation. Both paradigms together support an approach that brings the highest compromise between efficiency and (browser) compatibility.

2.3 Challenges in Implementation

As noted in earlier software projects mapping RDF based stores, a number of differences between RDF and Object Orientated design have been noted. [21] Among the problems identified include differences between class-based representation, structural inheritance and object conformance. In the first activity, we do not support a complete mapping but present a strictly controlled OWL-based representation generated in Protégé that allows for basic taxonomy definition and categorization with properties of the associate classes pointing to the data stores. Through maintenance of a data store closely conforming to O-O representation, we reduce potential problems with data mapping.
A second challenge is concerning the support of large data stores. In the case of referencing our OWL-based data store, client-heavy implementations can create scenarios involving very large data stores being constrained by memory limitations of the browser. To provide necessary error handling a method is provided to obtain the size of the data store (or subset) imported to the browser.
An additional issue is support of web services residing outside of the domain of the established server as well as applications that are not yet designed as REST-based services. These additional services are integrated through application of a CGI-based process. This implementation has been chosen over higher level security controls including use of automatic proxy generation which can add to complexities in implementation.

3 API

Our API is defined within three major categories in order to support the integration with the OWL data store, direct mapping of the selected OWL-S data stores and eventual invocation of the REST based web services.

3.1 Applied to the OWL Taxonomy

Javascript objects are allocated to support access to OWL-based data. setRes() allows the user to establish an OWL defined resource. This function returns an object providing methods to support the loading and subsequent discovery of SWS resources. The two main input arguments are the adapter and host. The assignment of individual objects provide support for multiple data sources.

```
var currentRes = setRes( adapter, host)
```

With currentRes assigned to a specified resource type, the data store access is loaded and enabled for SWS discovery activities including keyword match, query and reasoning statements.

```
currentRes.getRes(  Resourcename )
```

The size method defines the memory requirements of the data store to provide a means of error handing when loading very large data stores.

```
currentRes.size()
```

The find method provides keyword or basic expressions to be searched through the OWL class hierarchy.

```
currentRes.find( keyword_or_expression )
```

The query method allows for a data point format to be defined (such as SPARQL) and a query string to be applied.

```
currentRes.query( format, queryString )
```

Reasoning statements are performed by defining the format followed by a reasoning expression.

```
currentRes.reasoning( format, expression )
```

3.2 OWL-S Data Stores

Complete mapping of all four OWL-S based class definitions are performed. The function setOWLS accepts a URI for the services description class file and returns a Javascript object representing the data. The object , referenced by hashing in one of the class names (service, profile, process, grounding) allows for access to the defined data types.

var s = setOWLS(“uri")

3.3 Web Service Invocation

Using appropriate information from the OWL-S based definitions, web services are accessed from the WS object array by using the service name provided by an OWL-S description as a hash reference.

```
WS(“SWS_name”).refMethods()
```

Asynchronous callback functionality is provided in the context of the Javascript implementation through the means of the required naming conventions assumed by the wrapper. The naming convention requires that every callback method is defined as the method name followed directly by "_CB" as in example code below. In situations where HTML and 3D markup is shared via embedded object on the browser widows, asynchronous activity can be implemented without callbacks. In this case, efficiencies are provided by the use of return values from the REST WS calls thus eliminating unnecessary overhead.

```
WS(“SWS_name”).refMethods_CB()
```

3.4 Case Study

We demonstrate use of some of the functionality of the JAWS API through the development of an application to support numerical matchmaking of mathematical based services. This application involves the discovery, dynamic testing and invocation of mathematically based web services. [22][23] For this application we utilize a proprietary algorithm for the find method in which keyword searches are performed against an OWL-defined taxonomy of algorithms. In the first sequence of code we instantiate our object defining an OWL-based algorithm taxonomy.

```
var r =
setres(“adapter”,http://srl1xpm9q7h41.srl.ford.com)
r.getRes(“mathOnto.owl”)
r.size()
var arr = r.find(“optimization”)
```

The next segment of code presents the results of the find allowing for variables to be referenced from the OWL-S data stores to invoke a possible web service

```
for(I = 0;I = arr.length();i++){
      document.write( arr[i].className ,arr[i].childName,
      arr[i].URIproperty)
}
```

When a class is identified of interest, then its properties can be located via a method call (note the usage of 0 as subscript is arbitrary).

```
var s = setOWLs(arr[0].URIproperty)
```

Now call the web service by means of referencing the Javascript object with WS as the established naming convention for the data structure to contain references to the OWL-S defined web services. In our specific example there are two structures passed to the services.

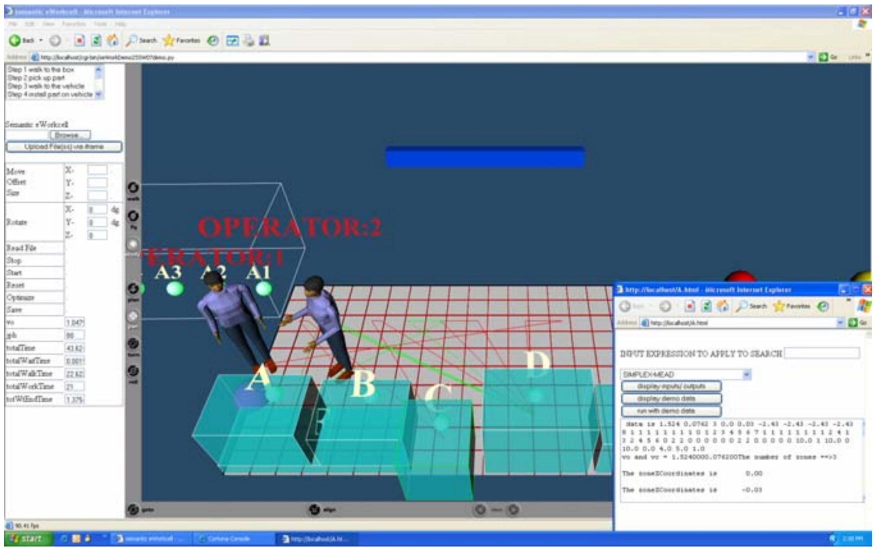
```
WS[“className”].optimization(obj_function, var_string)
```

Callback functionality is implemented by a user defining the callback function according to the set naming standard so it can be referenced by the JAWS API.

```
WS["className"].optimization_CB()
```

3.5 Use Case Scenario

A use case scenario demonstrating a subset of the functionality is illustrated in figure 2. In this case we utilize our API in a manufacturing simulation application.[24] This software allows for the design of workstation layouts in a manufacturing (assembly line) setting. The operator paths are dynamically generated via the definition of vehicle and operator velocities along with estimated task times, container location, zone location and associated synchronization activities. In this application we are interested in the incorporation of a web service based means of optimization to generate a suggested optimal design.
In this simulation software we incorporated our API to support a semi-automatic process of SWS discovery and integration. The controls of this portion of our integrated application are implemented via a pop-up window as shown in the lower right portion of figure 2. The first step in this application is to enter a keyword to be applied to the ontology search. This activity will in turn allow for a drop down menu to be populated with possible web service alternatives to be explored. Once a web service is selected, a user can display the input/output definition. Test data scenarios can also be executed to allow the user to examine the input/output requirements in order to trouble-shoot any integration issues.



```
 data is 1.524 0.0762 3 0.0 0.03 -2.43 -2.43 -2.43 -2.43
0 1 1 1 1 1 1 1 1 0 1 2 3 4 5 6 7 1 1 1 1 1 1 1 1 2 4 1
3 2 4 5 6 0 2 2 0 0 0 0 0 0 2 2 0 0 0 0 0 10.0 1 10.0 0
10.0 0.0 4.0 5.0 1.0
vo and vc = 1.5240000.076200The number of zones ==>3

The zoneXCoordinates is       0.00

The zoneZCoordinates is      -0.03
```

Figure 2 Manufacturing Simulation / Optimization incorporating JAWS API.

4 Conclusion

To support leveraging existing WS technologies, we have presented JAWS, a Javascript API for the purpose of SWS application development. This project breaks down the task of SWS development by creating a means to integrate existing open source implementations, mapping resultant data to Javascript objects thus decoupling integration from the rest of the processes in SWS. A prototype has been implemented allowing for keyword matching and application of SPARQL queries against an OWL ontology, mapping against OWL-S data stores and invocation of REST-based web services. A use-case scenario is presented in which a user is allowed to perform a semi-automatic process of switching in and out web services as necessary. Future work includes addition of the API to support multiple query languages and reasoners thus allowing server functionality to be further configured from a browser-based application.

Acknowledgements

Dr. Anton Eliens, Vrije Univeriteit, Amsterdam, The Netherlands for insight provided in developing this work.

References

1 Deitel, Harvey M., Deitel, Paul J., DuWaldt B., Trees L.K., Web Services: A Technical Introduction, Prentice Hall, 2002

2 Geromenko, Vladimir, Chen , Chaomei, Visualizing the Semantic Web: XML-based Internet and Information Visualization, Springer, 2005

3 J.Scicluna, C.Abela, M.Montebello,*Visual Modelling of OWL-S Services*, submitted at the IADIS International Conference WWW/Internet, Madrid Spain, October 2004

4 http://www.daml.org/services/owl-s/1.1B/owl-s.pdf

5 http://protege.stanford.edu

6. Chaiyakul, Sukasom, Limapichat, Kati, Dixit, Avani, Nantajeewarawat, Ekawit, "A Framework for Semantic Web Service Discovery and Planning, IEEE 2006

7 Srinvasan, Naveen, Paolucci, Massimo, Sycara, Katia, CODE: A Development Environment for OWL-S Web Services, 3rd International Semantic Web Conference ISWC2004

8 http://aws.amazon.com

9 http://code.google.com

10 http://www.daml.org/services/owl-s/1.1B/owl-s.pdf

11 Chase, Nicholas, "The Ultimate Mashup: Web Services and the Semantic Web", http://www-128.ibm.com/developerworks/edu/x-dw-x-ultimashup1.html

12. Zhou, Jiehan, Koivisto, Juha-Pekka, Niemela, Eila, A Survey on Semantic Web Services and a Case Study, IEEE Proceedings of the 10th International Coniference on Computer Supported Cooperative Work in Design, 2006

13. Dodds Leighh, "Introducing SPARQL: Querying the Semantic Web",http://www.xml.com/pub/a/2005/11/16/introducing-sparql-querying-semantic-web-

14. Carroll, Jeremy J., Reynolds, Dave, Dickinson, Ian, Seaborne, Andy, Dollin, Chris, Wilkinson, Keven, Jena:Implemetning the Semantic Web Recommendations, The 13th International World Wide Web Conference, 2004
15.Evren, Sirin, Bijan,Parsia, Bernardo,Cuenca Grau,Aditya, Kalyanpur and Yarden Katz, Pellet: A Practical OWL-DL resoner, Journal of Web Semantics, 2006
16..Babik, Marian, Hlucky, Ladislav, Deep Integration of Python with the Web Ontology Language, Scripting for the Semantic Web 2006
17 Oren, Eyal, Delbru, Renaud, Gerke, Sebastian, Haller, Armin, Decker, Stefan, "ActiveRDF: Object-Orientated Semantic Web Programming", WWW 2007, May-8-12, Banoff, Alberta Canada
18 D. Vrandecic, Deep Integration of the Scripting language and Semantic Web Technologies, Scripting for the Semantic Web 2005
19. Gross, Christian, "Ajax and REST Recipies, Apress 2006
20. Gehtland Justin, Galbraith Ben, Almaer, Dion, "Pragmatic Ajax: A Web 2.0 Primer", Pragmatic Bookshelf, 2006
21. Oren, Eyal, Delbru, ActiveRDF: object-orientated RDF in Ruby. Scripting for the Semantic Web, 2006
22. The MONET Consortium. MONET Architecture Overview, Technical Report Deliverable DO4, The Monet Consortium, March 2003 Available from http://monet.nag.co.uk
23. The MONET Consortium. The MONET Mathematical Query Ontology. Technical Report Deliverable D13, The MONET Consortium , March 2003, http://monet.nag.co.uk
24. Ostrowski, David A., A lightweight Framework for Web-Based, 3D, Information Visualization, Intnl. Conf. on Enterprise Inf. Systems and Technologies, EIWST 2007

Functional Programs as Linked Data

Joshua Shinavier
josh@fortytwo.net

Soph-Ware Associates, Inc.,
624 W. Hastings Rd, Spokane, WA 99218 USA
http://www.soph-ware.com

Abstract. The idea of linked programs, or *procedural* RDF metadata, has not been deeply explored. This paper introduces a dedicated scripting language for linked data, called Ripple, whose programs both operate upon and reside in RDF graphs. Ripple is a variation on the *concatenative* theme of functional, stack-oriented languages such as Joy and Factor, and takes a multivalued, pipeline approach to query composition. The Java implementation includes a query API, an extensible library of primitive functions, and an interactive command-line interpreter.

1 Introduction

Most of the data which populates today's Semantic Web is purely *descriptive* in nature, while the complex procedural machinery for querying, crawling, transforming and reasoning about that data is buried within applications written in high-level languages such as Java or Python, and is neither machine-accessible nor reusable in the Semantic Web sense. This paper explores the notion of *linked* or distributed programs as RDF graphs, and presents a functional, *concatenative* interpreted language, closely related to Manfred von Thun's Joy[1], as a proof of concept.

For a Turing-complete RDF query language, Ripple is an exercise in minimalism, both in terms of syntax and semantics. A Ripple *program* is a nested list structure described with the RDF collections vocabulary, and is thereby a first-class citizen of the Semantic Web. Significantly, the language is restricted to a single RDF query operation: the forward traversal of links. Broadly speaking, the goal of this project is to demonstrate that:

1. linked programs have all of the advantages of generic linked data[2]
2. a stack language is like a path language, only better
3. for most linked data purposes, forward traversal is all you need

[1] http://www.latrobe.edu.au/philosophy/phimvt/joy/j01tut.html
[2] http://www.w3.org/DesignIssues/LinkedData.html

The Java implementation resolves HTTP URIs, dynamically, in response to traversal operations, similarly to the Tabulator [1], the Semantic Web Client Library[3], and related tools[4]. Linked programs are drawn into the query environment in exactly the same manner, extending the the evaluation of queries to a distributed code base.

2 Syntax

Ripple's query model combines a computational scheme based on stack manipulation with a functional "pipes and filters" pattern. The stack paradigm makes for minimal, *point-free* syntax at the RDF level, while the *pipeline* mechanism accommodates RDF's multivalued properties by distributing operations over arbitrary numbers of intermediate results.

2.1 Textual Representation

The code samples in this paper are written in Ripple's own RDF syntax, which is very close to Turtle[5].
Note: throughout this paper, the namespace prefixes `rdf`, `rdfs`, `xsd`, `rpl`, `stack`, `stream`, `math`, `graph`, and `etc` are assumed to be predefined. Typing any of these prefixes, followed by a tab character, at the command line will reveal a number of terms in the corresponding namespace.

URIs may be written out in full or abbreviated using namespace prefixes.

```
<http://www.w3.org/2004/09/fresnel>.  # a URI reference
rdfs:Class.             # a qualified name
:marvin.                # using the working (default) namespace
rdfs:.                  # this URI has an empty local name
:.                      # legal, and occasionally meaningful
```

Keywords are the local names of a fixed set of special URIs. Currently, the local name of every primitive function is a keyword.

```
swap.                   # same as stack:swap
```

RDF Literals are represented as numbers and strings.

```
42.                     # an integer (xsd:integer for now)
3.1415926535.           # a floating point value (xsd:double)
"the Universe".         # a string Literal (xsd:string)
"English"@en.           # a plain Literal with a language tag
"2007-05-07"^^xsd:date. # a generic typed Literal
```

[3] http://sites.wiwiss.fu-berlin.de/suhl/bizer/ng4j/semwebclient/
[4] http://moustaki.org/swic/
[5] http://www.dajobe.org/2004/01/turtle/

Blank nodes lose their identity between sessions, but it's often useful to refer to them within a session.

```
_:node129n4ttifx2.       # a bnode with a generated id
_:tmp1.                  # a bnode with a user-defined id
```

Lists are indicated by parentheses.

```
("apple" "banana").      # a rdf:List
```

One symbol not found in Turtle is the *slash operator* (see Query Evaluation). Where it is affixed to an RDF property, it is equivalent to the forward traversal operator[6] of Notation3. A likely extension to the language will add quantifiers for *regular path expressions*, including `/?`, `/*`, and `/+`. E.g.

```
# @define smush: /*owl:sameAs.
```

2.2 Commands and Queries

The interface distinguishes between two kinds of statements: *queries*, which are expressions to be evaluated by the query processor, and commands or *directives*, which perform specific tasks. Currently supported directives include:

```
# Define a namespace prefix foo.
@prefix foo: <http://example.org/foo#>.

# Define :bar as the list (1 2 3).
@define bar: 1 2 3.

# Remove all statements about :bar
@undefine bar.

# Write (a bnode closure of) the terms in namespace foo: to a file.
@export foo: "file.rdf".

# Save the entire graph to a file.
@saveas "file.rdf".

# Quit the application.
@quit.
```

2.3 Compositional Syntax

At the level of lists and nodes, Ripple queries are expressed in postfix notation, or in *diagrammatic order*. Each query is a list in which the concatenation of symbols represents the composition of functions. When the functions are RDF properties, a query is equivalent to a path expression:

[6] http://www.w3.org/DesignIssues/N3Alternatives

```
# => "The RDF Vocabulary (RDF)"
("apple" "banana")/rdf:type/rdfs:isDefinedBy/dc:title.
```

This particular expression takes us from the list `("apple" "banana")` to its type, `rdf:List`, from the type to the ontology, `rdf:`, which defines it, and from the ontology to its title, `"The RDF Vocabulary (RDF)"`. The same idea applies to primitive functions:

```
# => recently pinged DOAP documents
10 "doap" /pingTheSemanticWeb/toString.
```

Here, too, we can imagine data (a *stream* of lists or *stacks*) flowing from the left hand side of the expression and emerging from the right hand side of the expression (the *head* of the stack) after undergoing a transformation of some kind. In this case, a stream of one stack containing the two arguments to the built-in "Ping the Semantic Web" function becomes a stream of ten stacks containing URIs which `toString` then converts to string literals.

2.4 RDF Representation

Every Ripple program is expressible in RDF, where it takes the form of a simple rdf:List. In the Java implementation, conversion between the RDF graph representation of a list and its more efficient linked list counterpart is implicit. For example, you may navigate a list using either list primitives or the RDF collections vocabulary, interchangeably.

```
("apple" "banana")/rdf:first.  # => "apple"
("apple" "banana")/uncons/pop. # => "apple"
```

When we assign a program a URI, we're pushing its definition to the same RDF model from which our query results are drawn:

```
@define hello:
    "Hello world!".
```

Given an appropriate base URI and web-visible triple store, the program itself becomes a part of the global graph of linked data, enabling remote users and applications to read it in, execute it, and build upon it without restriction.

3 Query Evaluation

Ripple's evaluation strategy hinges on a dichotomy between between `active` and `passive` stack items. Active items exhibit type-specific behavior, whereas passive items simply push a "copy of themselves" to the stack. Every item has a well-defined *arity* which, roughly speaking, is the number of arguments it consumes. To be precise, the arity is the depth to which the stack must be reduced before the item can be applied. For instance, the `swap` function requires two arguments, so the evaluator must make sure that the stack is normalized to two levels before `swap` receives it. Evaluation proceeds, *lazily*, until all active items have been eliminated from the head of the stack, at which point the stack is said to be in *normal form* (to one level). The only symbol active by default is the `rpl:op` operator, which has the effect of making the preceding item active as follows:

1. *RDF properties* consume a subject and map it to a stream of zero or more objects
2. *primitive functions* exhibit "black box", custom behavior
3. *lists* become an extension of the stack (execution "removes the parentheses")
4. *all other items* consume nothing and produce nothing (they just cause the stack to disappear)

Note: in the text notation, `rpl:op` is abbreviated as the slash prefix attached to the item it follows. For example, `(2 /dup)` is just a more compact way of writing `(2 dup rpl:op)`.

Now, consider the following definition and query:

```
# x => x*x
@define sq: /dup/mul.

# => 16
4/:sq.
```

The major steps in the evaluation of the query are as follows:

1. `(4 :sq rpl:op!)` – `rpl:op` is active by default
2. `(4 dup rpl:op! mul rpl:op!)` – dereference and dequote list `:sq`
3. `(4 dup rpl:op! mul!)` – `mul` primitive becomes active
4. `(4 dup! mul!)` – `mul` needs two arguments. Recurse
5. `(4 4 mul!)` – `dup` consumes its one argument and applies is rewrite rule
6. `(16)` – `mul` now has two reduced arguments, and applies its rule, yielding `16`

3.1 The Compositional Pipeline

Ripple differs from typical stack languages in that, rather than consuming a single stack as input and producing a single stack as output, Ripple's functions operate on *streams* containing any number of stacks. For instance, the following query yields not one, but several values:

```
@prefix foaf: <http://xmlns.com/foaf/0.1/>.
<http://www.w3.org/People/Berners-Lee/card#i>/foaf:knows.
```

If we compose it with another query, the second query needs to be capable of consuming not just a single stack, but many stacks, and of distributing its operation over all of them:

```
<http://www.w3.org/People/Berners-Lee/card#i>/foaf:knows/foaf:name.
```

To this end, Ripple conceives of functions as ”filters”, which behave like the elements of a pipeline: receiving input, transforming it, and passing it on. Filters may have state; for example, stream:limit filters (which count their input stacks and stop transmitting them after a certain point) or stream:unique filters (which remember their input stacks, and will not transmit a duplicate).

4 Examples and Use Cases

4.1 Arithmetic

In Ripple, as in Forth, stack shuffling operations take the place of bound variables in complex expressions.

```
# n => fibonacci(n)
@define fib:
    0 1 /rolldown     # push initial value pair and put n on top
    (/swap/dupd/add)  # push the step function
    /swap/times       # execute the step function n times
    /pop.             # select the low value

# => 13
7/:fib.
```

4.2 Recursion

Owing to the global nature of URIs, recursive definition is uncomplicated in Ripple. Functions may reference each other, and themselves, arbitrarily. Evaluation of @defined programs is delayed until forced by the evaluation of a query.

```
# n => n!
@define fact:
    /dup 0 /equal                 # if n is 0...
        (1 /popd)                 # yield 1
        (/dup 1 /sub /:fact /mul) # otherwise, yield n*fact(n-1)
    /branch.

# => 120
5/:fact.
```

4.3 Exploring a FOAF Neighborhood

The ability to request and aggregate RDF metadata on the fly gives Ripple's query engine the properties of a web crawler. ”Intelligent” applications aside, there are many conceivable use cases for simply discovering and aggregating a large chunk of data in a configurable fashion. The following program targets the decentralized, linked data of the FOAF network, beginning with Tim Berners-Lee's profile.

```
@prefix foaf: <http://xmlns.com/foaf/0.1/>.
@prefix owl:  <http://www.w3.org/2002/07/owl#>.

# foaf1 => foaf1, foaf2, foaf3, ...
@define foafStep:   # iterator for a FOAF crawler
    (   id          # include foaf1 itself
        owl:sameAs  # include nodes identified with foaf1
        foaf:knows  # include those foaf:known by foaf1
    )/each/i        # apply all three patterns at once
    /unique.        # eliminate duplicate results

# => names of TBL and friends, and of friends of friends
<http://www.w3.org/People/Berners-Lee/card#i>
    :foafStep 2/times /foaf:name.
```

4.4 Searching and Filtering in Revyu.com

Revyu.com is one of many[7] innovative web services which offer linked RDF views of their data. The web site links reviewers to reviews, reviews to things, and some things to book metadata in the RDF Book Mashup[8]. Here, we're more interested in narrowing the search space to a handful of "hits" than we are in the aggregated data as a whole.

```
# a f => a, if a/f is true, otherwise nothing
@define restrict:
    /dupd/i    # apply the filter criterion, f
        id     # keep the stack if a/f is true
        scrap  # throw the stack away if it isn't
    /branch.

@prefix scom: <http://sites.wiwiss.fu-berlin.de[no break]
/suhl/bizer/bookmashup/simpleCommerceVocab01.rdf#>.
@prefix foaf: <http://xmlns.com/foaf/0.1/>.
@prefix owl:  <http://www.w3.org/2002/07/owl#>.

# r => r, if r is a book review, otherwise nothing
@define bookReviewsOnly:
    (   /foaf:primaryTopic          # from review to topic
        /owl:sameAs                 # from topic to possible book
        /rdf:type scom:Book /equal  # is it really a book?
    )/:restrict.

@prefix rev:  <http://purl.org/stuff/rev#>.
```

[7] http://esw.w3.org/topic/TaskForces/CommunityProjects/LinkingOpenData/DataSets
[8] http://sites.wiwiss.fu-berlin.de/suhl/bizer/bookmashup/

```
# => labels of all of Tom's book reviews
<http://revyu.com/people/tom> /foaf:made
    /:bookReviewsOnly              # books only
    (/rev:rating 3 /gt)/:restrict  # 4 stars or better
    /rdfs:label.                   # from review to label
```

The query yields two results:

```
rdf:_1  ("Review of The Unwritten Rules of Phd Research, [...]")
rdf:_2  ("Review of Designing with Web Standards, by Jeff[...]")
```

Unlike the FOAF example, this program places a heavy burden on a single network server. If possible, it would be in the best interest of both server and client to offload query evaluation to the web service. Knowing when to use an API for federated queries, as opposed aggregating data, probably has more to do with the economics of distributed computing than with the rest of this paper. However, the API itself is certain to be a simple one. It could even be as simple as the passing of the dereferenceable URI of a single expression, to be resolved and evaluated remotely. Perhaps a REST service and an OWL-S service description could be generated based on a given query pattern.

4.5 Graph Transformations and Other Side Effects

The Ripple implementation includes a number of ”experimental” primitives which may be useful for graph transformations and in common metadata ”rewiring scenarios”[9]. As these primitives affect the state of their environment (for instance, by adding or removing statements), they are to be used with caution. The following is an example of a ”safe” application of the `new` and `assert` primitives. It transforms a resource description by creating a new blank node which retains and renames a few of the original node's edges.

Note: to see a description of these or any other primitive functions, type in their names at the command line, followed by a period.

```
@prefix foaf: <http://xmlns.com/foaf/0.1/>.

# node map => a new node with "mapped" edges
@define mapped:
    /each/i               # distribute over a list of pairs
    /new                  # create a new node
    i/dipd/rotate/assert  # assert a mapped statement
    1/limit.              # produce the node just once

# Maps to a minimal FOAF-like vocabulary.
@prefix ex: <http://example.org/minimalFoaf#>.
```

[9] http://simile.mit.edu/wiki/Rewiring_Scenarios

```
@define myMap:
    (rdf:type rdf:type)     # copy any rdf:type edges
    (foaf:name ex:name)     # map foaf:name to ex:name
    (foaf:knows ex:knows).  # map foaf:knows to ex:knows

# => minimal FOAF for TBL
<http://www.w3.org/People/Berners-Lee/card#i> :myMap/:mapped.
```

Other side effects may be more subtle. For instance, the following program affects the "Ping the Semantic Web" service which we queried earlier, possibly influencing subsequent query results.

```
# uri => uri (having pinged the Semantic Web)
@define ping:
    /toString/urlEncoding
    "http://pingthesemanticweb.com/rest/?url=" /swap/strCat
    /get.

# => PTSW's response to a ping of Ripple's DOAP URI/document
<http://fortytwo.net/2007/03/ripple/doap#>/:ping.
```

Other potentially useful side-effects include the sending of an e-mail or the firing of a system-specific event.

5 Implementation

Ripple is implemented in Java and uses the Sesame 2 (beta) RDF framework. The command-line interface relies on ANTLR for lexer/parser generation and on JLine for command history and tab completion. The project is built with Maven and is distributed under an open source license. Software releases are available at `http://fortytwo.net/ripple`.

6 Related Work

Ripple is strictly a resource-centric language and is not intended as an alternative to SPARQL. The fact that Ripple is rather like a path language makes it much more effective at some tasks than the "relational" SPARQL, and vice versa. The same can be said of any of the other RDF Path[10] languages, such as Versa[11] or the path portion of PSPARQL[12]. Probably the closest thing to Ripple is Ora Lassila's Wilbur[13] toolkit and path language, which integrates RDF with Common Lisp (coming soon: Python!). Deep integration [2] of languages such as Ruby [3] and Python [4] with RDF are related efforts, as well.

[10] http://esw.w3.org/topic/RdfPath
[11] http://copia.ogbuji.net/files/Versa.html
[12] http://psparql.inrialpes.fr/
[13] http://www.lassila.org/publications/2001/swws-01-abstract.shtml

7 Conclusion and Future Work

Ripple is an exploratory project, which is to say that further development will be driven by discoveries made along the way. If the hypertext web is any indication of the future of the web of data, it will be vast, complex, and overall, loosely structured. As it grows, we will need a sophisticated *web of programs* to keep pace with it.

In addition to the command-line interpreter, the distribution contains a query API, and the embedding of the Ripple query processor in further Semantic Web applications is a very likely use case. In the short term, I would like to integrate Ripple with a graphical RDF browser such as Longwell[14], and investigate the federated query scenario mentioned above.

Christopher Diggins has done some work on static typing for stack languages [5], and it would be interesting to see if or how such a type system can be expressed with an ontology. Currently, Ripple's type system is just a form of API documentation, so a more rigorous one would definitely be a step forward.

Compilation is another interesting possibility. While Ripple will always have an interpreted component, a modular Ripple program could be compiled to optimized Java bytecode and then reinserted into the environment as a primitive function.

References

1. Tim Berners-Lee et al. Tabulator: Exploring and Analyzing linked data on the Semantic Web. In Proceedings of the 3rd International Semantic Web User Interaction Workshop, 2006.
2. Vrandecic, D., Deep Integration of Scripting Languages and Semantic Web Technologies, In Soren Auer, Chris Bizer, Libby Miller, 1st International Workshop on Scripting for the Semantic Web SFSW 2005 , volume 135 of CEUR Workshop Proceedings. CEUR-WS.org, Herakleion, Greece, May 2005. ISSN: 1613-0073
3. Fernandez, O., Deep Integration of Ruby with Semantic Web Ontologies, see gigaton.thoughtworks.net/ofernand1/DeepIntegration.pdf
4. Marian Babik, Ladislav Hluchy: Deep Integration of Python with Web Ontology Language. Proc. of Scripting for the Semantic Web Workshop at the ESWC, Budva, Montenegro, June 12, 2006, CEUR Workshop Proceedings, ISSN 1613-0073, online CEUR-WS.org/Vol-181/paper1.pdf
5. Diggins, C., Typing Stack-Based Languages, available at http://www.cat-language.com/paper.html

[14] http://simile.mit.edu/wiki/Longwell

An Architecture to Discover and Query Decentralized RDF Data

Uldis Bojārs[1] and Alexandre Passant[2] and Frederick Giasson[3] and John Breslin[1]

[1] Digital Enterprise Research Institute, National University of Ireland, Galway
`[uldis.bojars, john.breslin]@deri.org`
[2] Université Paris IV Sorbonne, Laboratoire LaLICC, Paris, France
`alexandre.passant@paris4.sorbonne.fr`
[3] Zitgist LLC., Quebec City, Canada
`fred@fgiasson.com`

Abstract. In this paper we describe a distributed architecture consisting of a combination of scripting tools that interact with each other in order to help to find and query decentralized RDF data. Thanks to this architecture, anyone can participate in the collaborative discovery of Semantic Web documents by simply browsing the web. This system is useful for dynamic discovery of RDF content and can provide a useful source of RDF documents for other Semantic Web applications. Key components of this architecture are the Semantic Radar plugin used for discovery of Semantic Web data, Ping The Semantic Web service for aggregating locations of RDF data, and services using RDF data illustrated here with the example of doap:store.

1 Introduction

As the Semantic Web meme is spreading the number of RDF documents available on the Web is constantly growing and so is the number of tools creating and using this information. People create their FOAF[4] profiles, developers describe open-source projects using DOAP[5], and bloggers and bulletin boards provide data from their sites using SIOC[6] [4]. Semantic Web search engines such as Swoogle [6] and SWSE [10] have been developed to help to find this information, and browsers such as Tabulator [2] or Disco[7] can help to navigate through the Semantic Web.

The amount of documents indexed by Swoogle has reached more than 1.4 million but that is a small amount compared to the size of the Web estimated to be more than 11.5 billion documents [8]; the density of these RDF documents on the Web is still low and finding them can be a hard and laborious task. Moreover, an area which these search engines currently do not address is dynamic content, since information on dynamic websites such as blogs and online community sites is being generated at a fast pace. To adapt to this new trend, a new kind of infrastructure is necessary to enable Semantic Web applications to quickly harvest and use this information.

We present a scripting architecture that aims to solve this problem of finding and querying up-to-date decentralized RDF data, in almost real-time. Our goal is to show how simple scripting applications can work together to provide an extensible architecture for applications browsing and querying Semantic Web documents.

The rest of this paper is organized as follows. Section 2 describes our approach and its key features - decentralized scripting and user participation in discovery of Semantic Web data. Similar to the idea of Web 2.0 where tools and interfaces are

[4] `http://foaf-project.org/`
[5] `http://usefulinc.com/doap/`
[6] `http://sioc-project.org/`
[7] `http://sites.wiwiss.fu-berlin.de/suhl/bizer/ng4j/disco/`

driven by users the collaborative discovery provides a way to let everyone be part of this architecture. In section 3 we describe the complete architecture of the system consisting of the following components: (1) Semantic Radar, a browser extension used to detect presence of Semantic Web content while browsing the web, (2) Ping The Semantic Web (PTSW), a service for aggregating notifications about recently discovered and updated Semantic Web documents and (3) external services such as doap:store that use these data sources to provide browsing and querying interfaces. Finally, we will present a preliminary evaluation of our approach, introduce related work and conclude the paper by presenting some of the future work.

2 Our approach

A decentralized system. Using the philosophy of Unix programming, we decided to work on a set of small scripting application that each focus on a single job and try to do it as good as possible rather that working on a new centralized architecture. Thus, the system can be seen as a synergy of applications and web services assembled together in a "semantic pipeline": one tool will concentrate on finding documents, another on storing their URIs and providing them to consumers of RDF data, while the third will provide RDF query and browsing interfaces.

User participation. The other characteristic that differentiates this system from services such as Swoogle, is that it strongly involves user participation in the way we discover new Semantic Web documents. No preliminary knowledge of RDF or user effort is required, since users just have to browse the Web to be part of this discovery, as we will see in section 3.2. Thus, real users are involved in this architecture of participation and can help to discover the "invisible" Semantic Web.

Data provider. The centric point of our approach is the way to provide found data to users. Using previous discovery of Semantic Web content, but also pinged by other services such as TalkDigger[8], Ping The Semantic Web maintains a list of RDF documents URIs which is constantly updated, acts as a data provider and can be the entry point of a lot of services, as it provides fresh Semantic Web data.

Browsing and querying services. Using the list of URIs maintained by our data provider, external services that browse or query RDF data can be plugged to this architecture. The first implemented service of this kind is doap:store, a search engine dedicated to DOAP projects, described in detail in section 3.4, while other already existing services as SWSE are also using it.

3 Architecture of the System

Our system involves the following components (see Fig. 1):

- Data sources, i.e. RDF documents spread around the Semantic Web, created by people themselves or by some of the tools they are using as SIOC exporters[9];
- Semantic Radar[10], a Firefox plugin that allows anyone to be part of the discovering of Semantic Web documents by simply browsing the Web, using auto-discovery links to find RDF data from HTML pages;
- Ping The Semantic Web[11] (PTSW), a web service that stores a list of RDF document URLs it receives pings about, mainly via the Semantic Radar but also thanks to other services;

[8] http://talkdigger.com
[9] http://sioc-project.org/exporters
[10] http://sioc-project.org/firefox
[11] http://pingthesemanticweb.com

– Browsing and querying services, that use the list of documents URIs stored within PTSW, as doap:store[12] to query and browse DOAP projects or SWSE search engine.

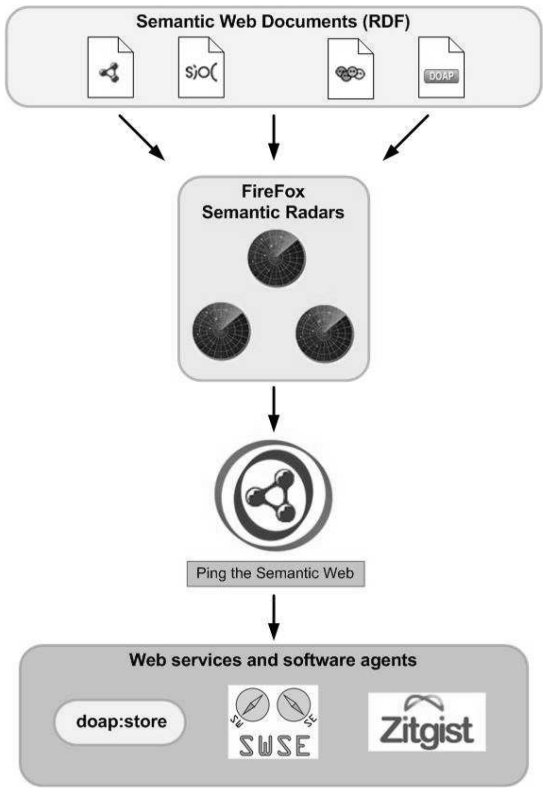


Fig. 1: Global architecture of the system

3.1 Data Sources and RDF Auto-discovery

The starting point of our architecture is RDF data sources that we need to discover. That includes all the RDF information published by people or information systems that they are using. Examples of RDF information on the web include SIOC, FOAF and DOAP data. An important question is how to detect links to RDF information from any web page.

RDF/XML Syntax Specification [1] recommends to use a `<link>` element in in the `<head>` element of HTML pages to point to additional RDF documents instead of directly embedding RDF/XML content in web pages, which may cause validation against DTD to fail, unless using RDFa or other embedded RDF approach. To use this technique the `href` element should point to the URI of RDF/XML content and the `type` attribute should contain the value `"application/rdf+xml"`.

```
<link rel="alternate"
  type="application/rdf+xml" title="RSS 1.0"
  href="http://apassant.net/blog/feed/rdf" />
```

This linking technique is known as RDF auto-discovery and is recommended and used by a number of Semantic Web projects and vocabulary specifications, such as

[12] http://doapstore.org

DOAP, FOAF[13], ICRA[14] and SIOC. The same technique with different MIME types is also widely used to point to RSS and Atom feeds. It offers automatic discovery of machine-processable information associated with webpages and applications are aware of that. I.e., web browsers use RSS auto-discovery links to display an RSS icon and to read or subscribe to RSS feeds associated with webpages.

By adding a link to FOAF profile a person enables visitors of the website to discover machine-readable FOAF data using web browser extensions such as Semantic Radar introduced in the next section. Data export tools such as WordPress SIOC plugin[15] use RDF auto-discovery links to facilitate discovery of the information they create.

It is its ease of understanding and implementing that makes RDF auto-discovery a popular choice for indicating presence of RDF data or any other metadata related to a web page.

3.2 Semantic Radar extension for Firefox

Semantic Radar is a Firefox browser extension (written in XUL and JavaScript) which inspects web pages for RDF auto-discovery links and informs a user about presence of them by showing icons in the browser's status bar.

When an auto-discovery link is detected, Semantic Radar examines the `title` attribute of the `link` tag. It uses this attribute as a hint - and only a hint, since it is designed for humans - and if its content matches a pattern associated with one of the data types the application is built to detect, it displays the corresponding icon. Currently supported data types are FOAF, SIOC and DOAP and the patterns used to detect them using the `title` attribute are simply "FOAF", "SIOC" and "DOAP".

This function makes users aware of the invisible Semantic Web that is part of web pages they are exploring every day. An important aspect of the tool is that it is not limited to a particular ontology (there were separate tools for detecting FOAF or other ontologies before that) but provides a generic way to detect various types of documents related to auto-discovery links and can be extended to cover more types of data in the future.

Semantic Radar allows users to click an icon and see this data in a user-friendly RDF browsing interface making it better understandable for human users. As such it can play a role in education and outreach of the Semantic Web to classic Web users and developers. By installing a simple browser plugin users can see what Semantic Web documents are associated with webpages and can explore this information without a need to look at raw RDF data. This is the first motivation for users to install this extension. E.g., when browsing a weblog that links to author's FOAF profile a user can get additional information about the author and his social relations.

The second function of Semantic Radar - "pinging" or sending notifications - is an important part of our architecture for collaborative discovery of Semantic Web data. Whenever a RDF auto-discovery link is discovered by the application it sends a "ping" to PTSW service - described in the next section - which collects and aggregates these notifications. As a result, users are building a "map" of Semantic Web documents, once again without additional effort. This provides a second motivation to install the plugin.

The ping function of Semantic Radar can be switched off at any time by clicking the radar icon in the status bar or via extension preferences thus addressing possible

[13] `http://rdfweb.org/topic/Autodiscovery`
[14] `http://www.icra.org/systemspecification/#specificLink`
[15] `http://sioc-project.org/wordpress`

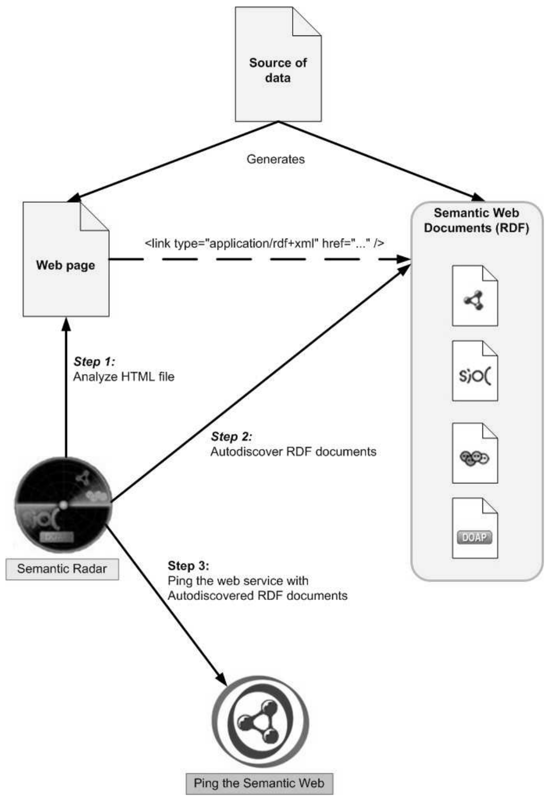

Source of data
Generates
Web page
<link type="application/rdf+xml" href="..." />
Semantic Web Documents (RDF)
Step 1: Analyze HTML file
Step 2: Autodiscover RDF documents
Step 3: Ping the web service with Autodiscovered RDF documents
Semantic Radar
Ping the Semantic Web
DOAP

Fig. 2: Overview of Semantic Radar

privacy questions. Additional safe-guards are provided by creating a *"black-list"* of URLs that pings will not be sent about, i.e., when browsing intranet pages.

Joint work of the users browsing the web using Semantic Radar extension and of PTSW service makes collaborative discovery of RDF documents possible. An important characteristic of this process is that the pings are generated by real users. Brin [5] uses an intuitive model of random walkers on the web to explain Google PageRank (the probability of a "random walker" visiting a web page is its PageRank). In collaborative discovery of information there are real "walkers" browsing the information that we can assume they are interested in. Thus the ping summary directly indicate not only presence of pages but also how popular they are.

3.3 Ping The Semantic Web Service

Ping The Semantic Web (PTSW) is a web service that acts as a multiplexer for RDF document notifications. It collects and archives notifications about recently updated or created RDF documents and HTML documents with RDF auto-discovery links, and provides an up-to-date list of Semantic Web documents to services (e.g., web crawlers, software agents) that request it. Similar web services are already widely used for aggregating changes to web feeds, with weblogs.com processing millions of pings each day. PTSW is dedicated to Semantic Web documents and all the sources they may come from: blogs, databases exported in RDF, hand-crafted RDF files, etc. This service can work together with the Semantic Radar or receive pings directly from content provides that let it know about new documents they produce. E.g., TalkDigger and revyu.com send pings for every RDF document they produce to ensure that the information is up-to-date even if nobody browses them using Semantic Radar.

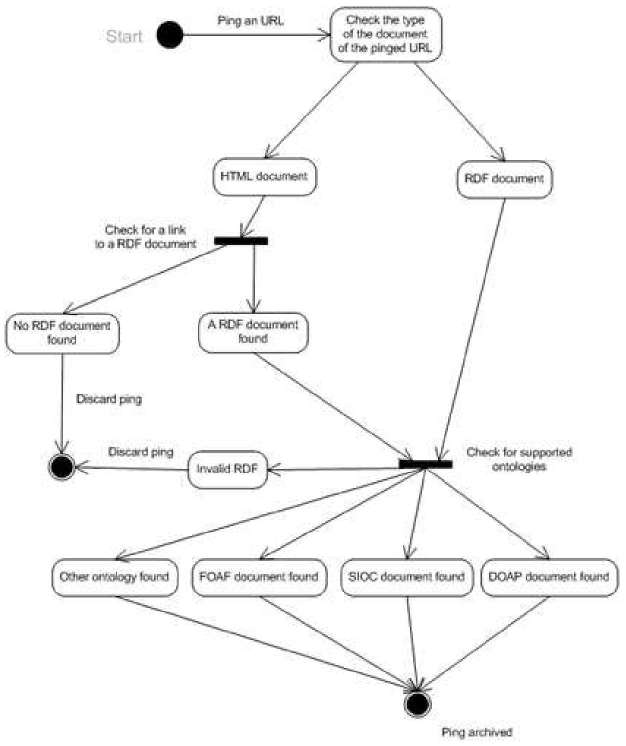


Fig. 3: Pings workflow in PTSW

Applications ping PTSW using one of the ping interfaces provided - REST [7] or XML-RPC. Once the service receives a ping and checks that the document is either RDF or HTML document with RDF auto-discovery links, it parses discovered RDF document(s) to categorize them in one or more of the following 5 categories based on presence of classes and properties from these ontologies: RDFS, OWL, SIOC, FOAF and DOAP. This list can be extended to other vocabularies in the future.

We decided to concentrate on FOAF, SIOC and DOAP first because they are widely available on the Web, and are created by end-users, content management tools and large websites such as FOAF from LiveJournal, SIOC from online community sites and TalkDigger, and DOAP from people and companies describing their software projects.

PTSW provides a live export of its list of URLs which can be used by software agents to find RDF documents. This list can be filtered according to type (vocabularies and ontologies used), date of update, RDF serialization, etc. The categorization of RDF documents by various types is used to help applications to get a list of documents they can understand and are interested in. This way we are minimizing the work of the software that requests a list, so that it can concentrate on manipulating the data instead of searching for it.

PTSW acts as one of the first steps of the Semantic Web food chain: Semantic Web search engines such as SWSE and Swoogle can use data from PTSW in order to quickly update recently changed RDF documents and to find next documents to index, using `rdfs:seeAlso` relationships that can exist from one document to another, and other third-party applications can use it to get an up-to-date list of RDF documents. Currently PTSW uses incoming pings to detect updated RDF documents. The service does not poll RDF documents for updates, but may be extended to do so.

3.4 doap:store

We use doap:store[16], a search engine dedicated to DOAP projects, throughout this paper as an example of a service that uses the list of RDF documents found by the Semantic Radar and PTSW. It is the first service to use PTSW as its main source of information. Since DOAP data are now created and used by many open-source developers and there was no easy way to find a project based on its DOAP description and metadata, we decided that the data collected by PTSW provide a good opportunity to write such a service.

Every hour a Python script gets the list of latest pings received by PTSW and categorised as DOAP documents. It parses the list of URLs, retrieves associated RDF documents and puts them in a local triple store, using the 3store API [9]. Since 3store supports contexts, projects can be easily updated when PTSW receives new pings: content of the old RDF file is removed from the store and replaced by the updated content, so that project descriptions are constantly up-to-date.

Contrary to existing software project directories such as Freshmeat[17], the main distinguishing feature of doap:store is its distributed approach. In regular project directories users have to register at a service and then describe their project using some specific forms, in doap:store they just need to publish a DOAP description of their project on the Web. Then they can ping PTSW directly or have a Semantic Radar enabled Firefox browser ping PTSW when they or visitors browse the webpage.

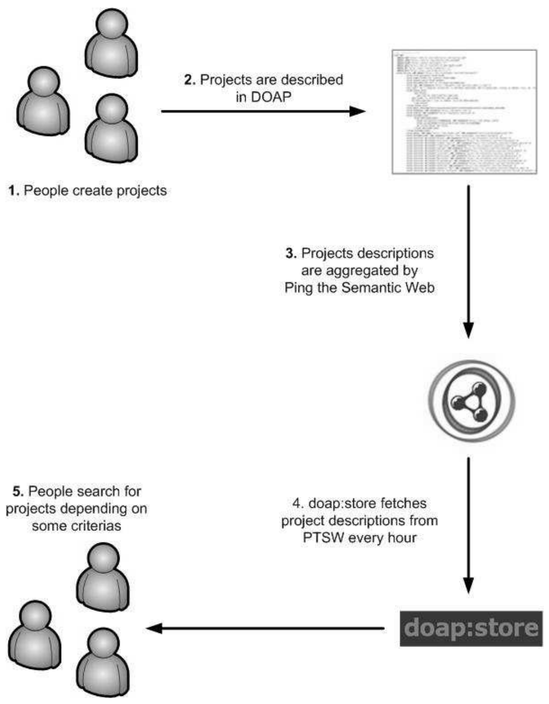

1. People create projects
2. Projects are described in DOAP
3. Projects descriptions are aggregated by Ping the Semantic Web
4. doap:store fetches project descriptions from PTSW every hour
doap:store
5. People search for projects depending on some criterias

Fig. 4: Information flow in doap:store

We believe that this approach shows what the Semantic Web can offer to both data providers and end-users as:

[16] http://doapstore.org
[17] http://freshmeat.net

- project maintainers just have to provide a single project description by creating it in a machine-readable format - DOAP - that can be understood by different tools such as doap:store and any generic RDF browser;
- they keep control of the project information since they don't need to publish it on various places where they will not be in control of their data anymore;
- end-users can get an immediate benefit of it, since updated description can be immediately seen in previously mentioned tools.

doap:store is written in PHP and is a quite a small application with about 600 line of code including HTML templates. One of the reasons that helped to realize it so quickly is that it does not have to deal with data discovery since this is managed by previous steps of the application chain introduced in this paper: Semantic Radar and PTSW. Therefore it only needs 10 lines of Python code to integrate and update data in its local database.

The interface offers various ways to query and browse documents:

- using a basic search engine, that allow to retrieve projects by name (`doap:name`), description (`doap:description` and `doap:shortdesc`), both of them, or by hostname (using the URI of the RDF graph corresponding to a `doap:Project`);
- using a tagcloud of programming languages. Tags are retrieved from DOAP files thanks to the `doap:programming-language` property, and then mashed-up to solve case variations and provide a case-insensitive tagcloud;
- using a YubNub[18] command line. Thus, doap:store can be queried with as simple queries as `doap name=rdf`, from web browsers search engines, or other YubNub tools;
- using SPARQL. For most advances users, a SPARQL endpoint is available, offering the way to query the data store or creating new documents from existing content of the data store using SPARQL `CONSTRUCT` instruction.

4 Evaluation

In order to evaluate our system, we made a short comparison of the number of files found by different search engines[19]. Swoogle identifies 416 documents using the DOAP ontology, while a Google search for `doap filetype:rdf` returns 448 documents. PTSW has matched 527 DOAP files, while SPARQL queries addressed to doap:store returns 446 document (using a graph query to find URIs that documents containing at least one instance of `doap:Project`) and 879 projects.

This comparison shows only data which are relevant to doap:store, i.e. a small amount of all RDF data available on the Semantic Web, but one important thing to notice is that this system provides fresh, almost real-time data. To illustrate this, the latest file indexed by Swoogle is from the mid-February 2007, while the latest new file registered in our system was retrieved at the beginning of April 2007. This is even more true with SIOC data since PTSW can record it as soon as data is created - if the blog engine adds it to the list of its pinging services.

PTSW currently has more than 7M documents indexed. A detailed evaluation of different types of data collected by Ping The Semantic Web is outside the scope of this paper and is planned for future work.

5 Related Work

Wiki pages, such as FOAFBulletinBoard[20], were one of the first tools to collect together a number of URIs of RDF documents. This approach works well while

[18] `http://yubnub.org`
[19] On March 30, 2007
[20] `http://rdfweb.org/topic/FOAFBulletinBoard`

there is only a small list of documents with every person adding one or two of them. Its disadvantages are: (1) these lists are usually not maintained (since it requires user intervention) and, as a result, loose quality over time; (2) it is not feasible to manually add URIs when many Semantic Web documents are being created at a fast pace, such as with blog data export in RDF.

SWSE and Swoogle are search engines for the Semantic Web. Swoogle [6] currently contains information on more than 1.4 million RDF documents and is used to find ontologies and Semantic Web documents. SWSE [10] crawls and indexes different types of web documents (XHTML, Atom, etc.), converts them to RDF and provides a user interface to help locate and navigate this information. They may provide APIs to access information indexed but the main use of these services is people using a web interface to query for data.

PiggyBank [11] is a Firefox browser extension which allows to collect RDF data in a distributed fashion. Harvest [3] is an early system that can be used to gather information from diverse repositories to build, search, and replicate indexes, and to cache objects as they are retrieved across the Internet.

Our architecture does not aim to replace Semantic Web search engines and concentrates on one task only - to discover RDF documents on the web and supply applications with a high quality list of Semantic Web documents. Its main use is to retrieve lists of RDF documents in a machine readable form using the API provided. An important difference from existing work is that a large number of users participate in a collaborative discovery of resources on the Web, and that a browser plugin is only the first step of a larger architecture.

6 Conclusion

In this paper, we described an architecture involving user participation in order to discover Semantic Web documents by simply browsing the Web, creating an up-to-date database of RDF documents which can be used by search engines and Semantic web applications to provide search and user-friendly services over this information.

We have shown how a combination of simple scripts can facilitate discovery of distributed Semantic Web data. Benefits of a number of independent applications acting together are that every application is good at a particular task and that new applications can be added to this pipeline because of open data formats used.

End-user applications such as doap:store can use a database of RDF documents provided and build user friendly applications. By bridging the gap between creators of Semantic Web data and the applications that use them, we expect this architecture to provide incentives to create more data and better applications, making the Web of Data into a reality.

While one of the principles of Web 2.0 is that tools and interfaces are driven by users [12], our framework realizes this by providing methods that allow everyone to be part of the Semantic Web initiative. This is a very important difference from blog ping aggregators because collaborative discovery involves and relies upon user participation - the architecture of participation in Web 2.0 terms. Through this architecture, we hope to see how user activity can lead to an enrichment of Semantic Web interfaces to distributed data.

7 Future Work

The future work regarding this architecture has two main aspects.

First, we will try to improve tools and architecture. For example, future versions of Semantic Radar should be able to let users define services other than PTSW they want to send pings to: that way, users could create their own pinging service that will make some specific actions when it receive a new ping. PTSW can also be improved, possibly including news types of data sources, such as SPARQL endpoints and metadata embedded in webpages. We also hope that other third party services using PTSW will be deployed, as we did with doap:store. Various search engines and front-ends can be created, such as the Zitgist search engine to be released soon.

Performance is not an issue for PTSW now but with a growing number of pings scalability can become a challenge for PTSW. We plan to address it by distributing work across a number of servers as required.

Next, second part will consist of more detailed analysis of data collected. Regarding collaborative discovery, an interesting case study would be to see if there is connections between navigation path and Semantic Web documents relationships. Thus, we could see if people browse HTML documents and follow paths that are related to the Semantic Web metadata they are associated to. To do that we will parse RDF documents registered by PTSW and extract connections between Semantic Web documents. We could also make further research about provenance of namespaces / classes per domain to identify various clusters of metadata.

8 Acknowledgements

This material is based upon works supported by the Science Foundation Ireland under Grant No. SFI/02/CE1/I131.

References

1. D. Beckett. Rdf/xml syntax specification. w3c recommendation, 2004.
2. T. Berners-Lee, Y. Chen, L. Chilton, D. Connolly, R. Dhanaraj, J. Hollenbach, A. Lerer, and D. Sheets. Tabulator: Exploring and analyzing linked data on the semantic web. In *Proceedings of the 3rd International Semantic Web User Interaction Workshop*, 2006.
3. C. M. Bowman, P. B. Danzig, D. R. Hardy, U. Manber, and M. F. Schwartz. The Harvest information discovery and access system. *Computer Networks and ISDN Systems*, 28(1–2):119–125, 1995.
4. J. G. Breslin, A. Harth, U. Bojars, and S. Decker. Towards Semantically-Interlinked Online Communities. In *The 2nd European Semantic Web Conference (ESWC '05), Heraklion, Greece, Proceedings*, May 2005.
5. S. Brin and L. Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Computer Networks*, 30(1-7):107–117, 1998.
6. L. Ding and T. Finin. Characterizing the semantic web on the web. In *5th International Semantic Web Conference*, 2006.
7. R. T. Fielding. Architectural styles and the design of network-based software architectures. *PhD dissertation,Dept. of Computer Science, Univ. of California, Irvine, Calif.*, 2000.
8. A. Gulli and A. Signorini. The indexable web is more than 11.5 billion pages. In *International World Wide Web Conference*, 2005.
9. S. Harris and N. Gibbins. 3store: Efficient bulk rdf storage, 2003.
10. A. Harth, J. Umbrich, and S. Decker. MultiCrawler: A Pipelined Architecture for Crawling and Indexing Semantic Web Data. In *5th International Semantic Web Conference*, 2006.
11. D. Huynh, S. Mazzocchi, and D. Karger. Piggy bank: Experience the semantic web inside your web browser. In *4th International Semantic Web Conference*, pages 413–430, 2005.
12. T. O'Reilly. What is web 2.0: Design patterns and business models for the next generation of software. *O'Reilly Nework*, 2005.

Empowering Moodle with Rules and Semantics

Sergey Lukichev, Mircea Diaconescu, Adrian Giurca

Brandenburg University of Technology at Cottbus
{Lukichev, M.Diaconescu, Giurca}@tu-cottbus.de

Abstract. This short paper describes preliminary ideas for empowering e-learning platform Moodle with rules and semantics. Many existing web applications already contain a lot of structured information, which is still not presented in machine-readable way. Extracting this information from an existing e-learning platform may give benefits to course tutors such as more control over course management, advanced reports and filters, reasoning over the course content. We describe how to represent the existing Moodle content in RDF and how to add rules on top of the RDF fact base. A semi-automatic method for rule mining and rule development is discussed.

Keywords: *Moodle, RDF, Semantic Web, Rule Mining, RDF-izing Moodle, e-learning, reasoning*

1 Introduction

In spite of a large amount of theoretical research conducted in the area of the Semantic Web, existing Semantic Web technologies are still hard to use practically and it is difficult for end users to take a benefit of them. The RDFa standard [3] is called upon to help web developers to enrich their content with semantics. There are debates and opinions that these standards are still difficult to employ and an alternative approach, using Microformats[1], claims about simplicity in introducing semantics to the existing web pages.

In order to get rid of the burden of the manual content annotation, a lot of work has recently been focused on extracting semantics from the existing content. Wikis, blogs, various content management systems and e-learning platforms contain a lot of unstructured data and this data as it is now cannot release the full power of its semantics. It cannot be shared using common vocabulary, there are no reasoning and semantic querying capabilities. An issue of extracting semantics from existing content becomes important and a number of works has been introduced for the semantic blogging [5] and for extracting semantics from the Wikipedia templates [4]. The Structured Blogging initiative[2] has developed plug-ins for the semantic annotation of Wordpress, MovableType and Drupal contents using Microformats.

[1] Microformats: http://microformats.org/
[2] Structured Blogging: http://structuredblogging.org

The use of Semantic Web technologies in e-learning may bring benefits to tutors and students. For instance, a tutor may want to know *which courses take those students, who usually take the Web Technology course?*, or *what is an average grade of students, who did not use a recommended course tutorial?* A traditional solution to answering such questions is to use relational databases (SQL queries, views). But moving to RDF is a shift to the open world with many distributed resources, identified by URIs as a mechanism for referring to global constants on which there exists some agreement among multiple data providers. Queries can be performed not only over a single database, but over the content of several distributed educational systems, including resources, which are externally available on the web.

As a working platform we use Moodle, an Open Source e-learning application, written in PHP and widely used by teachers all over the world. As a reasoning platform we choose Jena 2[3], a Semantic Web Framework, which allows reasoning on top of RDF fact bases with rules. Courses in Moodle already contain enough information to answer sample queries (see above) and many others, but in order to get answers, semantics should be extracted first and then rules on top of the structured data have to be defined.

We describe how to obtain an RDF fact base from the Moodle content and to empower it with rules. Rules add flexibility to the content analysis and give more control to tutors over the courses and teaching process. The distinct point of our approach among others is that it employs rules, which are obtained semi-automatically in two steps: *i)* using data mining methods over RDF for deriving initial rule base and *ii)* extending the initial rule base with more complex rules by a rule modeler (for instance, a course lecturer). The approach consists of the following steps:

1. Deriving the Moodle information model and generating an RDF fact base out of the Moodle content on the base of the information model (Section 2);
2. Mining rules, using the RDF fact base, obtained in the previous step. The mining process is automatic and its result is a rule base with rules in a form of $P(\overline{x}) \rightarrow Q(\overline{x})$, for example, *A student, who takes the Web Technologies course, usually takes the Logic for Information Systems course.* (See Section 3 for details);
3. Extending the initial rule base, obtained in the previous step with hand-crafted rules, which may contain additional conditions and define new concepts. Usually rules, defined on this step use logical atoms in their conditions, which have been obtained by the rule mining process. Rules on this step are created by tutors. For instance, *A student, who takes the Logic for Information Systems course and completes all assignments is a diligent student.* This is a derivation rule since it derives the concept of a *diligent student*, which may later be used in other rules or in querying. The condition part of this rule also contains the logical atom *A student takes a course*, defined in the step 2 by the rule mining process.

[3] Jena Semantic Web Framework: http://jena.sourceforge.net/

2 The Moodle Information Model

In this section we describe an excerpt from the Moodle information model (Figure 1), derived from the Platform Specific Model (PSM) of the Moodle database schema. The excerpt contains a number of classes and relations as a basis for RDF-izing existing Moodle content.

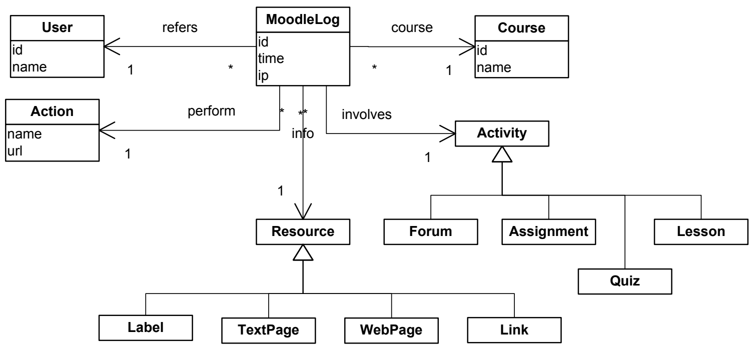


Fig. 1. An excerpt from the Moodle Information Model

The core class is the *MoodleLog* class, which represents a log entry in the Moodle database. All kind of actions, performed by students and tutors are recorded. Each record has one *User*, which may perform exactly one *Action* at a time over a specific *Resource* of a *Course*. A *Resource* is either a *Label*, a *TextPage*, a *WebPage* or a *Link*.

Every course may have a number of activities. Each *Activity* is either a *Forum*, an *Assignment*, a *Quiz* or a *Lesson*.

An RDF representation of this model is derived straightforward. Every UML class on the diagram corresponds to the RDF Schema class and every UML attribute/association corresponds to the RDF Schema property.

The RDF fact base is updated from the Moodle database automatically by a script, which runs periodically. Another option to maintain consistency between Moodle content and its RDF representation is an extension of Moodle, which generates RDF triples at the same time when the Moodle log is updated.

3 Rule Mining

The rule mining algorithms used here are provided by R. Agarwal et al in their seminal papers about mining association rules ([1], [2]). The *MoodleLog* class from the Moodle Information Model is a transactional database. A typical instance of this class (at the PSM[4] level) has the following values for its properties:

[4] Platform-Specific Model in the Model-Driven Architecture approach

```
((id,320834), (time, 1174041794), (userid, 219), (module, forum),
 (action, "view discussion"), (url, "discuss.php?d=413&parent=1350"))
```

These property-value pairs are translated into an RDF set of triples. As in the classical problem of mining association rules the goal is to generate all association rules that have *support* and *confidence* greater than the user-specified minimum support (minsup) and minimum confidence (minconf) respectively. Recall that the formal problem of mining association rules as in [1] is: Let $\mathcal{I} = \{i_1, \ldots, i_n\}$ a set of items. Let $\mathcal{D}$ be a set of transactions (i.e. our MoodleLog extension), where each transaction T is a set of items such that $T \subseteq \mathcal{I}$. Each transaction has a unique identifier (i.e. `id`). We say that a transaction T contains $X \subseteq \mathcal{I}$ if $X \subseteq T$. An association rule is an implication of the form $X \Rightarrow Y$ where $X \subseteq \mathcal{I}$, $Y \subseteq \mathcal{I}$, and $X \cap Y = \emptyset$. The rule $X \Rightarrow Y$ holds in the transaction set $\mathcal{D}$ with confidence c if $c\%$ of transactions in $\mathcal{D}$ that contain X also contain Y. The rule $X \Rightarrow Y$ has support s in the transaction set $\mathcal{D}$ if $s\%$ of transactions in $\mathcal{D}$ contain $X \cup Y$.

Applying the AprioriTid algorithm [2] on our RDF fact base generated from the Moodle logs, we can easily obtain a set of binary association rules involving RDF(S) triples. This is why the use of Jena 2 as a reasoning engine is appropriate. Notice that the algorithm shows best results on large data sets. Our actual Moodle log has more than 500000 entries. An example of a (typed) mined Jena 2 rule is:

```
(?X moodle:perform moodle:view) <-
        (?X rdf:type moodle:User) // type information
        (?X moodle:perform moodle:view_discussion)
        (moodle:view_discussion moodle:url "discuss.php?d=413&parent=1350"^^xs:string)
        (moodle:view moodle:involves moodle:quiz372)
        (moodle:quiz372 rdf:type moodle:Quiz) // type information
```

4 Extending the Rule Base

A rule base, obtained by the rule mining process (Section 3) can be extended with more complex rules, which may define new concepts, for instance, *A student, who has more than 10 forum posts is active in the forum discussion.*

```
(?X, userdef:activeInForum ?Forum) <- (?X rdf:type moodle:Student) (?X userdef:post ?Forum)
                                      (?X userdef:numberOfPosts ?Y) ge(?Y, "10"^^xs:integer)
```

This rule defines a concept of a student, who is active in the forum discussion. Such rules are created manually and a rule modeler should have a possibility to define rules in a user-friendly environment. One possible solution is an editor, which allows composing rules by non-technical users using a point-and-click interface with cut-and-paste pop-up windows and drop-down menus. Such interface allows selecting a predicate from a predefined list and filling term slots with values and variables. An example of such editor is provided in ILOG JRules[5]. Since the target rule platform is Jena 2, rules from the editor are translated into Jena 2 rules. This can be done using I1 rule interchange service, which allows rule interchange between different rule languages and platforms. The core of the

[5] ILOG JRules: http://ilog.com/products/jrules

interchange service is the R2ML loss-free rule interchange format ([7]), developed in the REWERSE Working Group I1[6].

Another possible solution is to use Attempto Controlled English[7] (ACE). ACE allows writing rules, using unambiguous subset of the English language. The work on the ACE to R2ML transformation (in order to obtain Jena 2 rules) has been started ([8], Chapter 1), but we consider the first solution with point-and-click interface as more appropriate for non-technical users.

5 Conclusion

In this paper we have described preliminary ideas for empowering e-learning platform Moodle with rules and semantics. Our further work on this project is towards practical implementation. The primary goal is to give tutors a useful application for flexible course management and reporting by implementing a Moodle module, based on the rule and semantics technologies such as RDF and Jena 2 and on Web 2.0 technologies as Ajax. The possible extension of the project is e-learning platforms integration, using Semantic Web Services, for instance, for rule interchange and querying of several fact bases.

References

1. R. Agrawal, T. Imielinski, A. N. Swami. Mining association rules between sets of items in large databases. In P. Buneman and S. Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 207–216, Washington, D.C., 26–28 1993.
2. R. Agrawal, R. Srikant, Fast algorithms for mining association rules. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *Proc. 20th Int. Conf. Very Large Data Bases, VLDB*, pages 487–499. Morgan Kaufmann, 1994.
3. B. Adida, M. Birbeck, RDFa Primer 1.0, 12 March 2007, http://www.w3.org/TR/xhtml-rdfa-primer/
4. Auer, S.; Lehmann, J.: What have Innsbruck and Leipzig in common? Extracting Semantics from Wiki Content. June 2007, Innsbruck, Austria
5. K. Mller, U. Bojars and J. G. Breslin. Using Semantics to Enhance the Blogging Experience. In 3rd European Semantic Web Conference (ESWC2006), Budva, Montenegro, June 2006.
6. G. Klyne and J.J. Caroll (Eds.), Resource Description Framework (RDF): Concepts and Abstract Syntax, W3C, 2004
7. G. Wagner, A. Giurca and S. Lukichev, Language Improvements and Extensions, REWERSE I1 Deliverable I1-D8, published as a technical report, March 2006, http://rewerse.net/deliverables-restricted/i1-d8.pdf
8. G. Wagner, S. Lukichev, N. E. Fuchs, Tool Improvements and Extensions 2, REWERSE I1 Deliverable I1-D11, published as a technical report, March 2007, http://rewerse.net/deliverables-restricted/i1-d11.pdf

[6] REWERSE I1 (Interchange services, downloads, etc): http://rewerse.net/I1

[7] Attempto Project Home Page: http://attempto.ifi.unizh.ch

Semantic Scripting Challenge Submissions

A user-friendly interface to browse and find DOAP project with doap:store
Alexandre Passant

Scripting a SIOC explorer
Benjamin Heitmann, Eyal Oren

Ripple: Functional Programs as Linked Data
Joshua Shinavier

A lookup index for Semantic Web resources
Eyal Oren, Giovanni Tummarello

Integrating SPARQL Endpoints into Directory Services
Sebastian Dietzold, Sören Auer

The challenge prize is kindly provided by:

T · · Systems

A user-friendly interface to browse and find DOAP projects with doap:store

Alexandre Passant

Université Paris IV Sorbonne, Laboratoire LaLICC, Paris, France
`alexandre.passant@paris4.sorbonne.fr`

1 Motivation

The DOAP[2] vocabulary is now widely used by people - and organizations - to describe their projects using Semantic Web standards. Yet, since files are spread around the Web, there is no easy way to find a project regarding its metadata.

Recently, Ping The Semantic Web[1] (PTSW) and Semantic Radar[2] plugin for Firefox introduced a new way to discover Semantic Web documents[1]: by browsing the Web, users ping the PTSW service so that it can maintain a contineously updated list of Semantic Web document URIs.

Thus, the idea of doap:store - `http://doapstore.org` is to provide a user-friendly interface, easily accessible for not RDF-aware users, to find and browse DOAP projects, using PTSW as a provider of data sources. This way, users do not have to register to promote a project as in freshmeat[3] or related services, but just need to publish some DOAP files on their websites to benefit of this distributed architecture. doap:store is the first implemented service using PTSW data sources to provide such browsing and querying features.

2 Architecture

doap:store involves 3 main components:

- A crawler: Running hourly, a tiny script parses the list of latest DOAP pings received by PTSW and then put each related RDF files into a triple-store;
- A triple-store: The core of the system, storing RDF files retrieved thanks to the crawler, and providing SPARQL capabilities to be used by the user-interface that is plugged on the endpoint;
- A user-interface: A simple interface, offering a list of latest retrieved projects, a case-insensitive tagcloud of programming languages, and a search engine to find DOAP projects regarding various criteria in a easy way.

While the crawler is written in Python, the interface is PHP5-based and the triple-store used is 3store[3], so both the crawler and the interface use its API to fetch and retrieve data. The whole application - without the API - is about 600 lines of code.

[1] `http://pingthesemanticweb.com`
[2] `http://sioc-project.org/firefox`
[3] `http://freshmeat.net`

3 Finding and browsing DOAP files

Apart the tagcloud used to find projects by programming language, a simple search-engine can be used to retrieve projects by (1) name (`doap:name`), (2) description (both `doap:desc` and `doap:shortdesc`), (3) name or description and (4) hostname (using the URI of the graph containing a project, since 3store is context-aware). A single SPARQL[4] query is used to find related projects, with an `FILTER REGEXP` expression added to the query depending on the search criteria.

Users can simply browse retrieved projects, ordered by name. Each project page provides a view of its available metadata, with links to the original RDF file and to a page displaying other projects from the same hostname. Since the DOAP ontology provides `rdf:label` for all its properties, not only project metadata but also property names are retrieved from the triple-store.

Another friendly way to query doap:store is to use YubNub[4], a command line service for the Web, since a `doap` command have been created for it. So, from their browser search engine or any YubNub client, users can type `doap desc=RDF` to be redirected to the doap:store results page listing projects with a description containing the `RDF` string.

Finally, for most advanced users, doap:store offers a SPARQL endpoint[5] - using a Javascript editor[6] - that can be used to query data or construct new RDF documents based on the actual content of the triple-store.

Thus, all these features provide various ways to retrieve informations about DOAP projects, from the easiest interface to the most advances SPARQL queries, in a single interface.

4 Acknowledgements

I would like to thank Uldis Bojārs and Frederick Giasson for their respective work on Semantic Radar and PTSW, that help to provide data sources for this service.

References

1. U. Bojārs, A. Passant, F. Giasson, and J. G. Breslin. An Architecture to Discover and Query Decentralized RDF Data. In *3rd Workshop on Scripting For The Semantic Web (SFSW07)*, June 2007.
2. E. Dumbill. DOAP: Description of a Project. http://usefulinc.com/doap/.
3. S. Harris. SPARQL query processing with conventional relational database systems. In *International Workshop on Scalable Semantic Web Knowledge Base System (SSWS 2005).*, 2005.
4. E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. W3C Working Draft, W3C, 2006. http://www.w3.org/TR/rdf-sparql-query/.

[4] `http://yubnub.org`
[5] `http://doapstore.org/sparql.php`
[6] `http://dannyayers.com/2006/09/27/javascript-sparql-editor`

SFSW Challenge Entry: Scripting a SIOC explorer

Benjamin Heitmann and Eyal Oren

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

Exploring SIOC data In order to explore SIOC data from multiple online social communities in an integrated way, we have developed a SIOC explorer. It enables importing and exploring SIOC data from community sites if they expose their content by publishing it in the SIOC [1] format. The SIOC explorer can be found online[2], and its source code can be found at the launchpad site[3].

Using the SIOC explorer The entry page shows a list of SIOC forums in the database. In terms of SIOC each collection of posts is called a "forum". After selecting a forum, a list of post excerpts is shown in the main column. The user can expand a specific post in order to see the full content and comments. The user can then browse posts by author or by topic or by period of time across all forums in the database. On the left side a possible list of filters is displayed, which can be used to restrict which posts get displayed.

Benefits for the user Firstly, different types of community sites, like weblogs, forums, mailing lists and IRC chat logs can export their content as SIOC data. This includes not only posts but also replies, information about authors and commenters and the links between all entities. The SIOC explorer uses this information to give the user a unified view on the content and structure of multiple sources.

Secondly, the SIOC format allows for richer content metadata. Topics can refer to terms from a SKOS[4] taxonomy, person references can point to FOAF[5] profiles, post titles and creation timestamps are expressed with the Dublin Core[6] vocabulary. The SIOC explorer allows the user to explore and navigate the data using all the available metadata.

In contrast, RSS[7] based news readers can only aggregate posts, and they can only use the limited metadata capabilities of RSS, like author names and category keywords.

[1] `http://rdfs.org/sioc/spec/`
[2] `http://www.activerdf.org/sioc/`
[3] `http://launchpad.net/sioc-ex`
[4] `http://www.w3.org/TR/swbp-skos-core-spec/`
[5] `http://xmlns.com/foaf/0.1/`
[6] `http://dublincore.org/`
[7] `http://web.resource.org/rss/1.0/`

Capabilities of the Semantic Web Besides allowing shared concepts between different sources, the Semantic Web allows each source to use different and evolving schemata to describe new concepts and to mix vocabulary from different ontologies. SIOC data can refer to SKOS topic descriptions or to FOAF personal profiles. In the SIOC explorer this gives the user the ability to filter posts based on fine-grained constraints:

Example of a complex type: FOAF maker When browsing a forum with more then one author, each of the authors is identified by an instance of `FOAF:maker`. When browsing the posts of such a forum, the user can locate the filter “maker” in the left column, click on “show details” and then select e.g. the workplace of a specific maker instance. Only posts from the maker with the specified workplace will then be displayed.

Browsing data from foreign schemata The SIOC explorer can handle data using vocabularies, which are not associated with SIOC data, because the navigation engine is domain agnostic and only relies on the features of RDF data. We use this feature to put e.g. the week of each post in the database using `SIOCEX:week`. SIOC posts only have a timestamp, but materialising this property using our own vocabulary allows the user to easily browse posts by week, month or year. This can be seen in the left column after selecting a forum.

Usage of a Scripting Language To develop the SIOC explorer, we extended the Ruby on Rails framework[8] with components for consuming and processing Semantic Web data. The first component is ActiveRDF[9], which addresses the “model” mismatch and maps RDF data onto objects. The second component is BrowseRDF[10], a faceted browsing and navigation engine that enables exploration of large Semantic Web datasets without domain-specific knowledge. The third component is a SIOC crawler which crawls, extracts, normalises and integrates SIOC data.

Each of the three components is designed to augment and integrate with Ruby on Rails. ActiveRDF can serve as a data layer in Ruby on Rails, replacing or augmenting the default ActiveRecord layer. The BrowseRDF navigation algorithms are implemented as a library for Ruby on Rails and provides generic navigation on top of ActiveRDF. The SIOC crawler uses several libraries and command-line tools which are external to Ruby, and incorporates the results into the model of the Ruby on Rails application.

Using ActiveRDF and our other extensions, the integration of Rails with RDF data was straightforward and the development effort was quite low compared to the benefits of using Semantic Web data. The models itself are automatically provided as virtual models, the controller (with all application logic) contains around 95 lines of code and the views contain around 100 lines of abstract HTML. The SIOC crawler consists of around 150 lines of code.

[8] `http://www.rubyonrails.org/`
[9] `http://www.activerdf.org/`
[10] `http://www.browserdf.com/`

Ripple: Functional Programs as Linked Data

Joshua Shinavier
josh@fortytwo.net

Soph-Ware Associates, Inc.,
624 W. Hastings Rd, Spokane, WA 99218 USA
http://www.soph-ware.com

Abstract. Ripple is a scripting language expressed in RDF lists. Its scripts both operate upon and are made up of RDF metadata, extending the idea of HTTP dereferenceability to computation. Ripple is a variation on the "concatenative" theme of functional, stack-oriented languages such as Joy and Factor, and distinguishes itself through a multi-valued, "pipeline" approach to query composition, as well as the inherent distributability of its programs. The Java implementation of Ripple includes a query engine, a provisonal assortment of primitive functions, and an interactive interpreter which parses commands and queries in a readable, Turtle-like format. A demo application can be found at: http://fortytwo.net/ripple.

Key words: RDF, scripting language, semantic web, linked data

1 Linked Data

"Linked data"[1] is the subset of the Semantic Web which associates statements about the "thing" identified by a particular URI with the corresponding web location. An appropriate HTTP request for such a URI should produce an RDF document containing statements about it. Ripple assumes, furthermore, that the set of statements in such a response is *complete*, and the software may reject statements about the URI from any other source. This restriction makes the language referentially transparent with respect to its one RDF query operation, forward traversal.

2 RDF Equivalence

Not only does Ripple *operate* on linked data, but its programs are meant to *be* linked data as well. Every Ripple expression is equivalent to a collection of type rdf:List. In the Java implementation, conversion between the RDF graph representation of a list and its more efficient linked list counterpart is transparent to the user application. RDF properties are identified with functions which map subjects to objects. For instance, in the following query, `dup`, `toString`,

[1] http://www.w3.org/DesignIssues/LinkedData.html

`sha1`, `swap`, and `equal` are all primitive functions, whereas `foaf:mbox` and `foaf:mbox_sha1sum` are properties:

```
@prefix :      <http://fortytwo.net/2007/03/ripple/demo#>.
@prefix foaf:  <http://xmlns.com/foaf/0.1/>.
@define goodMboxSha1Sum:
    /dup /foaf:mbox/toString/sha1 /swap /foaf:mbox_sha1sum /equal.
```

The `@define` directive creates a new list and identifies it with a URI, in this case: `http://fortytwo.net/2007/03/ripple/demo#goodMboxSha1Sum`. The list may then be applied as a function:

```
<http://www.w3.org/People/Berners-Lee/card#i>/:goodMboxSha1Sum.
```

The above is a program which dereferences Tim Berners-Lee's FOAF URI, finds the sha1 sum of his email address, and checks it against the stated value, yielding `true`. Characteristically of Ripple, it doesn't matter where or when a program is executed; provided that its URI is made to be dereferenceable (for instance, by exporting data to a public location, or eventually, by attaching the interpreter to a triple store with a SPARQL endpoint), the program itself is dereferenceable, and a Ripple query engine on a remote machine will draw it into its own execution environment as needed.

3 The Compositional Pipeline

If Joy[2] is a stack language, then Ripple is a "stream-of-stacks" language. Each of its functions consumes a series of stacks as input, and produces a series of stacks as output, which is how it gets away with using RDF properties as functions. This simple query, for instance, yields not one, but several values:

```
<http://www.w3.org/People/Berners-Lee/card#i>/foaf:knows/foaf:name.
```

To distribute operations over an arbitrary number of values, Ripple replaces functions with "instances" which behave like elements of a pipeline: receiving input, transforming it, and passing it on. Instances may have state; for example, instances of the `limit` primitive (which counts its input and stops transmitting them after a certain point) or of the `unique` primitive (which remembers its input, and will not transmit a duplicate stack).

4 Conclusion and Future Work

Ripple is an exploratory project, which is to say that further development will be driven by discoveries made along the way. Thus far, Ripple has been most useful for quickly picking out and exploiting useful patterns in linked data. If the hypertext web is any example, the web of data will be vast, complex, and overall, loosely structured. As it grows, we will need an equally sophisticated web of programs to keep pace with it.

[2] http://www.latrobe.edu.au/philosophy/phimvt/joy/j00ovr.html

Scripting challenge entry: A lookup index for Semantic Web resources

Eyal Oren and Giovanni Tummarello

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

Finding Semantic Web statements The Semantic Web can be seen as a large knowledge-base of statements about resources, forming an interconnected graph through multiple references to the same resources. But the graphs in the Semantic Web are decentralised: there is not one single knowledge base that contains the graph of statements but instead anyone can contribute statements on his "personal" web-space. The complete graph is only visible after crawling and integrating the fragments mentioned on these personal subspaces. For developers of Semantic Web applications, which operate on Semantic Web data, the decentralisation poses a challenge: how and where to find statements (information) about certain resources?

A simple lookup index We have developed (an initial version of) a simple lookup index called Sindice that helps developers in this situation. Sindice is available online[1], the code is available open-source[2] under the LGPL license.

Benefits for the user By itself our index has no benefits for the end-user. But any application that uses Semantic Web data, such as the Disgo[3], Tabulator[4], or BrowseRDF[5] RDF browsers, the DBin[6] information sharing client, or the SIOC browser[7], can use Sindice to offer *their* users a better service. Next to every resource that they encounter these applications can place a "find more information". When users click that button, their application would request a list of relevant sources with more information about a resource, and follow one or more of these sources, learning things that others said about that particular resource. Instead of crawling the Semantic Web themselves, they need only ask Sindice for pointers to good sources.

[1] http://activerdf.org/sindice
[2] http://launchpad.net/sindice
[3] http://sites.wiwiss.fu-berlin.de/suhl/bizer/ng4j/disco/
[4] http://www.w3.org/2005/ajar/tab
[5] http://browserdf.com
[6] http://dbin.org
[7] http://activerdf.org/sioc

Crawling sources Users can request explicit crawls of their updated documents through the RESTful API[8]. Apart from that, Sindice periodically (currently every twenty minutes) requests recently updated documents itself from http://pingthesemanticweb.com and visits each source to index its contents. Once our hardware infrastructure is stable we will also index large RDF dumps from Swoogle[9], SWSE[10] and Wikipedia[11]. With our current datastructure, and based on our currently indexed data, we estimate that storing 300 million resources (which is our minimal target) will occupy approximately 77Gb of data, which is quite readily available on ordinary hardware.

Ranking results We rank results using a cheap and potentially "useful enough" algorithm. We give precedence to sources on the same hostname as the queried resource (following the linked-data model that resources should have meaningful descriptions on their own de-referenceable URIs. We further rank sources on their PageRank, their size (in amount of statements) and their amount of statements. We also investigate a feedback model where often-used sources would be boosted as well.x

Project scope Sindice is a simple lookup index and its scope is limited: it does not index the actual RDF in the documents but only the resources mentioned; it does not allow full queries on the data but only lookups from resource to mentioning sources. Keeping the scope focused we are able to keep our index small and fast, namely an on-disk hashtable: `resource` $\Rightarrow$ `source[]`.

Implementation Sindice is built using the Ruby scripting language for rapid prototyping and uses the Ruby on Rails Web application framework to handle routing of requests and offering a RESTful API with various response formats (HTML, JSON, and XML). Sindice uses several external libraries such as the Redland[12] "rapper" RDF parser and the QDBM[13] persistent hashtable. Sindice also uses several online services such as http://pingthesemanticweb.com, which is queried periodically to find recent RDF documents, and an estimation[14] of Google's PageRank to estimate relative importance of sources. Apart from the automatically generated Ruby on Rails code, the core Sindice library is implemented in around 200 lines of code (including comments) and the Web application, handling requests and generating HTML, JSON, and XML responses in around 130 lines of code (including comments).

---

[8] http://activerdf.org/sindice/parse/URL
[9] http://swoogle.umbc.edu
[10] http://swse.deri.org
[11] http://dbpedia.org
[12] http://librdf.org
[13] http://qdbm.sourceforge.net
[14] http://seopen.com/seopen-tools/pagerank.php

Integrating SPARQL Endpoints into Directory Services

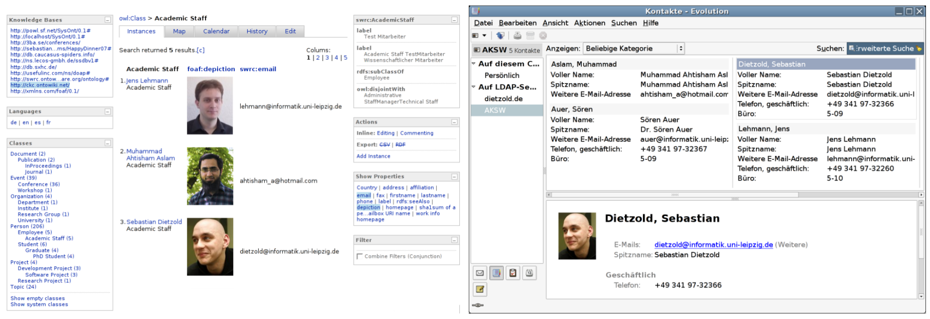
Sebastian Dietzold[1] and Sören Auer[1,2]

[1]University of Leipzig, Germany
Department of Computer Science
`dietzold@informatik.uni-leipzig.de`

[2]University of Pennsylvania, USA
Department of Computer and Information Science
`auer@seas.upenn.edu`

We demonstrate[1] the integration of RDF knowledge bases from our social, semantic collaboration tool OntoWiki via SPARQL endpoints into directory services based on the Lightweigth Directory Access Protocol (LDAP). In order to achieve this, we translate LDAP queries into SPARQL queries and transform the incoming SPARQL results back into the LDAP data model. The transformation component is implemented as a backend for the widely used OpenLDAP server[2].

LDAP based directory services are an important part in the IT infrastructure of most organisations and enterprises. They act as a central service for integrating new applications into an IT infrastructure and can be accessed by many different types of clients ranging from content management systems to personal email tools.


Knowledge Bases
Languages
de | en | es | fr
Classes
owl:Class > Academic Staff
Instances
Map
Calendar
History
Edit
swrc:AcademicStaff
Actions
Inline: Editing | Commenting
Export: CSV | RDF
Add Instance
Show Properties
Country | address | affiliation | email | fax | firstname | lastname | phone | label | rdfs:seeAlso | depiction | homepage | sha1sum of a pe...ailbox URI name | work info homepage
Filter
Combine Filters (Conjunction)
Show empty classes
Show system classes
Kontakte - Evolution
Datei Bearbeiten Ansicht Aktionen Suchen Hilfe
AKSW 5 Kontakte
Auf diesem C...
Persönlich
Auf LDAP-Se...
dietzold.de
AKSW
Anzeigen: Beliebige Kategorie
Suchen: :rweiterte Suche

http://powl.sf.net/SysOnt/0.1#
http://localhost/SysOnt/0.1#
http://3ba.se/conferences/
http://sebastian...ms/HappyDinner07#
http://db.caucasus-spiders.info/
http://ns.lecos-gmbh.de/ssdbv1#
http://db.sxhc.de/
http://usefulinc.com/ns/doap#
http://swrc.ontow...are.org/ontology#
http://ckc.ontowiki.net/
http://xmlns.com/foaf/0.1/

Document (2)
Publication (2)
InProceedings (1)
Journal (1)
Event (39)
Conference (36)
Workshop (1)
Organization (4)
Department (1)
Institute (1)
Research Group (1)
University (1)
Person (206)
Employee (5)
Academic Staff (5)
Student (6)
Graduate (4)
PhD Student (4)
Project (4)
Development Project (3)
Software Project (3)
Research Project (1)
Topic (24)

Search returned 5 results.[c]

Colums: 1 | 2 | 3 | 4 | 5

Academic Staff	foaf:depiction	swrc:email
1. Jens Lehmann Academic Staff		lehmann@informatik.uni-leipzig.de
2. Muhammad Ahtisham Aslam Academic Staff		ahtisham_a@hotmail.com
3. Sebastian Dietzold Academic Staff		dietzold@informatik.uni-leipzig.de

label
Test Mitarbeiter
label
Academic Staff TestMitarbeiter Wissenschaftlicher Mitarbeiter
rdfs:subClassOf
Employee
owl:disjointWith
Administrative StaffManagerTechnical Staff

Aslam, Muhammad
Voller Name: Muhammad Ahtisham Asl
Spitzname: Muhammad Ahtisham Asl
Weitere E-Mail-Adresse ahtisham_a@hotmail.com

Auer, Sören
Voller Name: Sören Auer
Spitzname: Dr. Sören Auer
Weitere E-Mail-Adresse auer@informatik.uni-leip:
Telefon, geschäftlich: +49 341 97-32367
Büro: 5-09

Dietzold, Sebastian
Voller Name: Sebastian Dietzold
Spitzname: Sebastian Dietzold
Weitere E-Mail-Adresse dietzold@informatik.uni-l
Telefon, geschäftlich: +49 341 97-32366
Büro: 5-09

Lehmann, Jens
Voller Name: Jens Lehmann
Spitzname: Jens Lehmann
Weitere E-Mail-Adresse lehmann@informatik.uni-
Telefon, geschäftlich: +49 341 97-32260
Büro: 5-10

Dietzold, Sebastian
E-Mails: dietzold@informatik.uni-leipzig.de (Weitere)
Spitzname: Sebastian Dietzold
Geschäftlich
Telefon: +49 341 97-32366

Fig. 1. "Same same but different": Visualization of a common RDF model in OntoWiki (left) and over LDAP in the email-reader evolution (right)

Typical content which is stored in directory services is information about users or hardware. Even though it has a complex schema with global unique

[1] Sources and documentation: http://aksw.org/Projects/LDAP/Backend.
[2] http://www.openldap.org

object identifiers and sophisticated syntax matching rules, LDAP lacks powerful semantics and is restricted to a tree like data model.

To combine the user acceptance of LDAP directory services and the semantic power of RDF knowledge bases, we decided to implement a SPARQL endpoint backend for OpenLDAP.

The core functionality of the backend is:

1. *Transformation of given LDAP query strings into SPARQL queries*: This is done in a straightforward way. LDAP query strings allow AND, OR and NOT composition of simple attibute filters. An attribute filter consists of an attribute name, a filter type (equality, presence, order, ...) and a filter value. The corresponding SPARQL query uses UNION, OPTIONAL, FILTER and BOUND constructs and matched schema names.
2. *Match RDF schema / OWL ontology URIs to LDAP schema identifiers*: There are two ways to deliver RDF data over LDAP directories. First, it is possible to translate a complete LDAP schema to OWL [2, 3]. In the following, this OWL ontology can be used for instance creation. Second, we allow to configure matching tables for easy integration of given RDF models (e.g. a mapping from FOAF[3] to inetOrgPerson [4]).

This demo will allow users to modify and query the content of an RDF knowledge base[4]. The modifications can be done using the semantic collaboration tool OntoWiki [1]. To query the knowledge base, any LDAP capable mail user agent can be used (see figure 1). The demo is not limited to OntoWiki knowledge bases but can be used in combination with arbitary SPARQL endpoints containing information about people.

References

1. Auer, S., Dietzold, S., Riechert, T.: OntoWiki - A Tool for Social, Semantic Collaboration. In Cruz, I.F., Decker, S., Allemang, D., Preist, C., Schwabe, D., Mika, P., Uschold, M., Aroyo, L., eds.: The Semantic Web - ISWC 2006, 5th International Semantic Web Conference, ISWC 2006, Athens, GA, USA, November 5-9, 2006, Proceedings. Volume 4273 of Lecture Notes in Computer Science., Springer (2006) 736–749
2. Dietzold, S.: Generating RDF Models from LDAP Directories. In Auer, S., Bizer, C., Miller, L., eds.: Proceedings of the SFSW 05 Workshop on Scripting for the Semantic Web , Hersonissos, Crete, Greece, May 30, 2005. Volume 135 of CEUR Workshop Proceedings., CEUR-WS (2005)
3. Dietzold, S.: Basic vocabulary to use LDAP data in RDF. OWL ontology (2005) http://purl.org/net/ldap.
4. Smith, M.C.: Definition of the inetOrgPerson LDAP Object Class. RFC 2798, The Internet Engineering Task Force (IETF) (2000) http://www.ietf.org/rfc/rfc2798.txt.

[3] http://www.foaf-project.org/

[4] We want to use the model of all submitted ESWC2007 FOAF documents here.