SWAD-Europe Deliverable 4.1: Semantic Web and Web Services: RDF/XML and SOAP for Web Data Encoding

Project name:
W3C Semantic Web Advanced Development for Europe (SWAD-Europe)
Project Number:
IST-2001-34732
Workpackage name:
4. Semantic Web and Web Services
Workpackage description:
☞http://www.w3.org/2001/sw/Europe/plan/workpackages/live/esw-wp-4.html
Deliverable title:
Semantic Web and Web Services: RDF/XML and SOAP for Web Data Encoding
URI:
☞ http://www.w3.org/2001/sw/Europe/reports/xml_graph_serialization_report
Authors:
Dan Brickley, Max Froumentin
Abstract:
This document reports on work in SWAD-Europe to develop, test and evaluate techniques that integrate RDF/XML and SOAP Encoding approaches to Web data interchange. The main focus is on RDF/XML and SOAP Encoding as alternative XML data syntaxes for a common edge-labeled graph data model. RDF tools can make use of SOAP Encoding documents (typically used when exchanging protocol messages with a Web service) by treating SOAP Encoding as an alternative syntax for the RDF graph data model. By using a prototype mapping between the SOAP Encoding Data Model and RDF, implemented as an XSLT stylesheet, we can aggregate SOAP Encoded messages with other data formats that map to RDF.

STATUS:
The RDF/SOAP work is complete, and this document is in final revisions for the 4.1 deliverable. It will be maintained and continued throughout the life of the larger workpackage, in the light of implementation experience with deliverable 4.2 (Semantic Web Services demo).

Intended Audience: this document addresses a number of related technology issues from the fields of Semantic Web and Web Service standardisation. For brevity, familiarity with the basics of SOAP, XML Namespaces and RDF is assumed.

Contents

Introduction: SOAP Encoding and the Web

Introduction - This document reports on the first phase of the SWAD-Europe Web Services / Semantic Web work, concentrating on the comparison of SOAP and RDF's graph data structures. It provides an account of the SOAP Encoding Data Model in terms of edge-labeled graphs, URI references and the XML Namespaces specification. To test this model, it also proposes a mapping of this abstract model to a simple text format (N-Triples) for quality assurance and testing for SOAP Encoding implementations. This is compared to existing unit test frameworks for SOAP toolkits, and the application of this approach to data aggregation is demonstrated using an RDF-based object-relational query application. This provides context for a discussion of the goals and technical issues relating to a proposed mapping between the SOAP Encoding Model and W3C's RDF data format.

The SOAP 1.2 specification from W3C's XML Protocol Working Group (XMLP) includes a "SOAP Encoding" framework that can be used to serialize and de-serialize complex data structures in XML. The SOAP Encoding is widely used in Remote Procedure Call (RPC) applications to provide a language and platform-neutral representation of programmatic data structures. SOAP toolkits typically offer object marshalling and unmarshalling facilities which exploit the SOAP Encoding simplify the creation of Web-based applications and services.

This document explores the details of the data model and its wider applicability to Web data and service aggregation. Although typically deployed in a transitory, protocol-oriented (messaging) context, the SOAP encoding is an XML document format (like SVG, XHTML, MathML) in its own right, and may prove useful in non-protocol non-RPC (and non-protocol) contexts. If this is the case, it is important to clarify any subtleties in the specification of the SOAP Encoding Data Model, since the assumptions shared by RPC-oriented applications may not be shared by other users of the specification. To this end, a SOAP Encoding test case format is proposed, based on an alternative non-XML serialization of the SOAP Data Model.

A (tentative) account of the SOAP Encoding Data Model is presented below, followed by a worked example showing the concrete encoding of this data model in XML. The model is presented first, and without examples, to illustrate the difficulty of abstracting an account of the Data Model in isolation from its intended use and XML

representation.

The SOAP Encoding Data Model - The SOAP Encoding Data Model is a subset of the Directed Labeled Graph (DLG) class of data models. As such it provides a simple type system that is a generalization of common features found in type systems in programming languages, databases and semi-structured data. Edges in the graph are labeled with simple ("locally scoped") string values or ("globally scoped") URI-qualified names corresponding to the XML namespace mechanism. Nodes in the graph are either simple (datatyped) values or represent complex objects that are described by an aggregation of relations (labeled edges) to other values. Node datatyping as represented using URI references (for example to the XML Schema datatypes).

The SOAP Encoding Data Model abstracts away from the details of the referencing mechanisms (such as XML ID/IDREF, URI, XML element containment) that support any particular concrete encoding of a SOAP data graph. SOAP Encoding Data Model instances, regardless of encoding format, can be merged to aggregate the encoded information by folding together the abstract edge-labeled data graphs. Globally scoped (URI-named) edges and nodes provide one of many strategies to support SOAP data model merging. A node in the SOAP Encoding Data Model graph structure is said to be "URI-ref" labeled if it has an type of xsd:URIRef. SOAP graph merging provides one strategy to support Web service composition, by providing a common representation in which information fragments acquired from multiple Web serves can be combined.

The SOAP data model provides an abstract view of XML-encodable data graphs, represented as an unordered set of node-edge-node 3-tuples. XML serializations of the SOAP data model are by necessity ordered. All concrete XML encodings of the SOAP data model must therefore specify the significance of document (Infoset) ordering with respect to the abstract edge-labeled graph model. Within a compound value (complex node), each relation to another node is potentially distinguished by a role name, ordinal or both. In this latter case, the edge name itself corresponds to a pair of ordinal and role name. Datatyping information can be reflecting in the abstract graph structure through 'type' edges from literal nodes, with the 'type' pointing from some literal data to a node (typically URI-labeled) that represents the datatype of the literal node.

The SOAP data model is consistent with, but does not specify, a number of mechanisms for describing a "DLG Schema" corresponding to meta-information about node and edges types in the SOAP data graph. UML, RDF and other representational formalisms may provide additional information about the node and edge types that occur in a SOAP data model instance. SOAP does not mandate the use of any particular mechanism for this. DLG-schema information can be used to support the concrete serialization of SOAP data graphs into XML, for example by providing information about cardinality or domain and range constraints on (globally scoped) edge types.

The SOAP Encoding syntax - The SOAP Encoding Data Model can more easily be presented through the use of examples that show SOAP Encoding deployed in SOAP 1.2 XML protocol messages. The goal of this document is to draw a distinction between SOAP Encoding *as typically used* and SOAP Encoding *as it could be used*. So we start with SOAP Encoding as typically used...

The Encoding format represents ideas from a number of contexts, but is most closely aligned with the RPC tradition. It is frequently used to translate programmatic object (instances of Java classes, Perl, C#, Ruby, Python etc.) into a more neutral form for communication to other Web applications. Often these objects will themselves be transitory things, representations of structured information stored in relational or object databases. Since the SOAP Encoding uses XML Schema datatypes, datatyped information can be mapped between database, software and XML representations with relative ease. Most SOAP tools seem to focus on the software side of things rather than directly hooking SOAP Encoding up with the 'raw' information sources (RDBMS etc). (The examples shown here will be elaborated on later to show how SOAP Encoding might have a natural mapping to database structures too.)

Worked Example: Ruby Application Archive (RAA) SOAP service

These examples are based on some experimental implementation work using the Ruby programming language. As such it is fitting to draw our examples from the Ruby community. The Ruby Application Archive is a Web-accessible listing of software packages for the Ruby language. It is available online as HTML documents, and a SOAP interface is also available, offering a variety of ways to access the data.

Our example takes a single 'record' from the RAA, and shows its representation in Ruby, and in the SOAP Encoding wire format. We begin with the XML form, encoded using the SOAP Data Model Encoding of SOAP 1.2:

RAA Record in "on the wire" SOAP Encoding format

So what do these records look like on the wire? Here's a dump from a logged SOAP protocol session (note that we are using SOAP 1.1 not the later 1.2 spec due to the scheduling of this deliverable; the basic approach hasn't changed greatly). This is the XML that gets sent through the network (possibly via intermediary SOAP nodes):
(examples/soap-eg1.xml.esc)

```
<?xml version="1.0" encoding="utf-8" ?>

<env:Envelope xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

<env:Body>

<n2:getInfoFromNameResponse
    env:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:n1="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:n2="http://www.ruby-lang.org/xmlns/soap/interface/RAA/0.0.1">

<return xsi:type="n2:Info">

  <update xsi:type="xsd:dateTime">2001-06-18T08:43:46Z</update>

  <owner xsi:type="n2:Owner">
    <email xsi:type="xsd:string">hd@rubylinks.de</email>
    <id xsi:type="xsd:string">hd@rubylinks.de-Hans-Dieter Stich</id>
    <name xsi:type="xsd:string">Hans-Dieter Stich</name>
```

```
  </owner>

  <product xsi:type="n2:Product">
    <download xsi:type="xsd:string">
http://www.rubylinks.de/download/ruby-spread-0.1.tar.gz</download>
    <homepage xsi:type="xsd:string">http://www.rubylinks.de/</homepage>
    <status xsi:type="xsd:string">just started, first alpha release</status>
    <description xsi:type="xsd:string">Ruby/Spread is an interface to
    the Spread Library at
    http://www.spread.org/. Spread is a toolkit and daemon that
    provides multicast and group communications support to
    applications across local and wide area networks. Spread is
    designed to make it easy to write groupware, networked
    multimedia, reliable server, and collaborative work
    applications.
     </description>
     <version xsi:type="xsd:string">0.1</version>
     <license xsi:type="xsd:string">LGPL</license>
     <name xsi:type="xsd:string">Ruby/Spread</name>
   </product>

  <category xsi:type="n2:Category">
   <minor xsi:type="xsd:string">Distributed Communication</minor>
   <major xsi:type="xsd:string">Library</major>
  </category>

</return>

</n2:getInfoFromNameResponse>

</env:Body>

</env:Envelope>
```

What happens at the other end, when this XML is received? Some SOAP library turns this Encoding back into objects that a programmer can work with (perhaps manipulate, show to a user, or store in a database, or use in conjunction with other objects/data from elsewhere in the Web. The receiving application needn't be written in the same language as the original service, but the SOAP library will provide some mapping between the Encoding format and a representation that is more convenient to work with. Our example here shows one mapping from the SOAP Encoding Data Model into Ruby classes and objects. SOAP 1.2 facilitates, rather than specifies, such mappings.

RAA record in Ruby

This is the data structure we're ultimately interested in. It defines classes and properties, and is implemented using the Ruby-SOAP marshalling system.

Ruby code:

```
module RAA
   InterfaceNS = "http://www.ruby-lang.org/xmlns/soap/interface/RAA/0.0.1"   class Category
    include SOAP::Marshallable
    @@typeNamespace = InterfaceNS     attr_reader :major, :minor
                        def initialize( major, minor = nil )
      @major = major
      @minor = minor
    end     def to_s
      "#{ @major }/#{ @minor }"
    end     def ==( rhs )
      if @major != rhs.major
       false
      elsif !@minor or !rhs.minor
true
      else
@minor == rhs.minor
      end
    end
  end   class Product
    include SOAP::Marshallable
    @@typeNamespace = InterfaceNS     attr_reader :name
    attr_accessor :version, :status, :homepage, :download,
                :license, :description
    def initialize( name, version = nil, status = nil, homepage = nil,
                download = nil, license = nil, description = nil )
      @name = name
      @version = version
      @status = status
      @homepage = homepage
      @download = download
      @license = license
      @description = description
```

```
    end
  end   class Owner
    include SOAP::Marshallable
    @@typeNamespace = InterfaceNS     attr_reader :id
    attr_accessor :email, :name     def initialize( email, name )
      @email = email
      @name = name
      @id = "#{ @email }-#{ @name }"
    end
  end   class Info
    include SOAP::Marshallable
    @@typeNamespace = InterfaceNS     attr_accessor :category, :product, :owner, :update
                def initialize( category = nil, product = nil, owner = nil, update = nil )
      @category = category
      @product = product
      @owner = owner
      @update = update
    end
  end   Methods = {
    'getAllListings' => [ 'Array' ],
    'getProductTree' => [ 'Hash' ],
    'getInfoFromCategory' => [ 'Array', 'category' ],
    'getModifiedInfoSince' => [ 'Array', 'time' ],
    'getInfoFromName' => [ 'Info', 'name' ],
  }
end
```

RAA Record: example Ruby instance

Here is an actual instance of this Ruby class, pretty printed using the Ruby inspect() method, with indenting added to show the structure:

```
<RAA::Info:0x401eba18
  @product
       = #<RAA::Product:0x401e9880
           @download
               ="http://www.rubylinks.de/download/ruby-spread-0.1.tar.gz",
           @homepage
               ="http://www.rubylinks.de/",
           @description
               ="Ruby/Spread is an interface to the Spread
       Library at http://www.spread.org/. Spread is a toolkit and
       daemon that provides multicast and group communications
       support to applications across local and wide area
       networks. Spread is designed to make it easy to write
       groupware, networked multimedia, reliable server, and
               collaborative work applications.",
           @version="0.1",
           @status="just started, first alpha release",
           @license="LGPL",
           @name="Ruby/Spread"
  >,
  @category
     = #<RAA::Category:0x401e823c
         @major="Library",
         @minor="Distributed Communication"
 >,
 @update
    = #<Date: 105929806913/43200,0,2299161>,
 @owner
    =#<RAA::Owner:0x401eaaa0
         @id = "hd@rubylinks.de-Hans-Dieter Stich",
         @email="hd@rubylinks.de",
         @name="Hans-Dieter Stich"
 >
>
```

Ruby instance as classes and properties

This is pretty straightforward so far. From a programmer's perspective (in Ruby, but would be very similar in Java, Python, Perl, ...), we have the following classes and properties:

- class **RAA:Info** has properties: product, category, update
- class **RAA:Product** has properties: download, homepage, description, version, status, license, name
- class **RAA:Category** has properties: major, minor
- class **RAA:Date** has a datatyped value
- class **RAA:Owner** has properties: id, email, name

For example, the value of the 'update' property of our RAA:Info object is an object of type RAA:Date; the value of the 'product' property is an object of the (similarly named) class RAA:Product. This code follows the common

convention of using initial capitals for class names.

The SOAP Encoding Data Model revisited: a common data structure

So what do these various representations have in common? Broadly, that the can be thought of as representing data structured in a directed labeled graph, with objects as the nodes, and labeled arcs connecting them. Each triple of node/arc/node that makes up the graph must somehow be exported using XML's syntactic constructs (angle brackets, element names, attribute names, namespaces, datatyping). The role of the SOAP Encoding portion of the specification is exactly that: to account for the detail of how we write down XML descriptions of these graph structures in a way that is well defined and easily processed.

SOAP Encoding (diagram): RAA record as edge-labeled graph

The data model can be seen quite clearly when represented diagrammatically:
(☞SVG version)

Notes and Qualifications

This diagram does not include datatyping information, although the full (see below) N-Triples representation of the graph would include a number of 'type' edges connecting the literal data nodes to nodes that stand for their datatype. The exact treatment of the date node is unmotivated: why did we use a 'value' arc rather than make the content the label for the typed date node? In short, because it looked like the date was a proper class and not just a datatyping convention.

Graphs and Trees: motivating the SOAP Encoding Graph Data Model

Note that our current example is very simple, since there are no loops in the graph. In the general case SOAP Encoding can be used to serialize arbitrary edge labeled graphs, not just trees. The SOAP Encoding syntax uses XML's ID and IDREF mechanism to achieve this. This maps well onto the needs of RPC applications, since objects may have references to objects that have already been serialized.

For now we focus on a simple case; the full utility of the SOAP Encoding graph Data Model only really becomes apparent when we explore the composition of multiple SOAP-based Web services through merging partial information from multiple sources. For example, consider a second SOAP Web service that contains "white pages" information about the individuals named in the records described by the RAA Web service, or a third SOAP Web service that contained detailed package management information (dependency charts) or documentation for the software packages described by RAA. Web services will only reach their full potential when multiple such services can be cheaply combined across the Web. To achieve this, we need mechanisms such as the SOAP Encoding Data Model that provide a general purpose abstraction into which complementary chunks of data can be mapped.

Discussing the SOAP Encoding Data Model in the abstract is hard, and discussing it in terms of the SOAP 1.2 serialization *rules* for encoding that model in XML can be confusing. The approach adopted here is to focus on the abstract model, made concrete in a simple test cases format called 'N-Triples' and illustrated graphically using "node and arc" diagrams.

This has two benefits. Firstly, we acquire a test cases format for SOAP Encoding implementations that, critically, is *independent* from the details of the RPC applications which have to date been the main users of the syntax. Secondly, we can explore use of a format-agnostic graph representation that allows us to experiment with mapping SOAP Encoding instance data into other formalisms such as RDF. This is explored further in the final sections of this document.

Focussing on the abstract graph structure (via the N-Triples data dump syntax) allows the wider Web community to explore the value of this data format, since it makes no assumptions about intended use. The SOAP Encoding model could be processed through a SOAP-specific API, mapped directly into object/relational database structures, or (as is common) used to create programmatic objects (instances of Java classes etc.). A data format for testing SOAP Encoding implementations needs to be neutral across all such applications, since SOAP itself is intended to provide a neutral platform for integrating these diverse environments.

Adopting the N-Triples test case format - One technique for ensuring interoperability between applications that use SOAP Encoding is to test that some objects (Java etc) can be turned into XML and reconstructed again. This works pretty well for RPC-oriented applications. The approach explored here is intended to complement that approach, providing a basis for data-oriented applications to test their interpretation of the SOAP Encoding component of SOAP 1.2.

☞N-Triples is a simple, line oriented text representation of the abstract data graph. It was created by the RDF Core Working Group to represent test cases for RDF/XML syntax, which serves a similar role in clarifying the RDF syntax specification and testing the performance of RDF parsers. The working hypothesis here (see Appendix A for implementation and notes) is that the SOAP Encoding and RDF graph data models are close enough for N-Triples to be applied for use with the SOAP Encoding.

An N-Triples document has a tabular structure. Each row (line in a text file) corresponds to a node-edge-node construction in a SOAP Encoding Data Model graph. The three columns stand for the object reference, property type, and actual value for that data fragment. Since all SOAP Encoding instances can be decomposed into object/property/value triples corresponding to this graph structure, we can use simple text files as a test case format for representing the content carried by the SOAP Encoding XML syntax. In practice, there are some complications (which we'll come to later).

N-Triples example: RAA record

Aside: mapping to RDF

Note that the use of the RDF namespace for the edge labels "type" and "value". The first prototype was produced using RDF-based tools, and it wasn't clear which namespace to use for these constructs (XML Schema ns, the SOAP Encoding ns, or RDF's). Since N-Triples is based on NTriples from RDF Core, we currently use some RDFisms. Our primary goal here is not, however, to map the SOAP Encoding Data Model to RDF's, but to explore the utility of that Data Model and some tools for testing adherence to it. Mapping to RDF (to RDF's graph model, or to specific graph constructs from the RDF world) is a means to an end, not an end in itself.

Object ID	Property type	Property value

<_:o1>	<http://example.ruby-language.org/ rda-vocab#owner>	<_:o5> .
<_:o5>	<http://example.ruby-language.org/ rda-vocab#name>	"Hans-Dieter Stich" .
<_:o5>	<http://example.ruby-language.org/ rda-vocab#email>	"hd@rubylinks.de" .
<_:o5>	<http://example.ruby-language.org/ rda-vocab#id>	"hd@rubylinks.de-Hans-Dieter Stich" .
<_:o5>	<http://www.w3.org/ 1999/02/22-rdf-syntax-ns#type>	<http://example.ruby-language.org/ rda-vocab#Owner> .
<_:o1>	<http://example.ruby-language.org/ rda-vocab#update>	<_:o4> .
<_:o4>	<http://www.w3.org/ 1999/02/22-rdf-syntax-ns#value>	"2001-06-18T08:43:46,0Z" .
<_:o4>	<http://www.w3.org/ 1999/02/22-rdf-syntax-ns#type>	<http://example.ruby-language.org/ rda-vocab#Date> .
<_:o1>	<http://example.ruby-language.org/ rda-vocab#category>	<_:o3> .
<_:o3>	<http://example.ruby-language.org/ rda-vocab#minor>	"Distributed Communication" .
<_:o3>	<http://example.ruby-language.org/ rda-vocab#major>	"Library" .
<_:o3>	<http://www.w3.org/ 1999/02/22-rdf-syntax-ns#type>	<http://example.ruby-language.org/ rda-vocab#Category> .
<_:o1>	<http://example.ruby-language.org/ rda-vocab#product>	<_:o2> .
<_:o2>	<http://example.ruby-language.org/ rda-vocab#name>	"Ruby/Spread" .
<_:o2>	<http://example.ruby-language.org/ rda-vocab#license>	"LGPL" .
<_:o2>	<http://example.ruby-language.org/ rda-vocab#status>	"just started, first alpha release" .
<_:o2>	<http://example.ruby-language.org/ rda-vocab#version>	"0.1" .
<_:o2>	<http://example.ruby-language.org/ rda-vocab#description>	"Ruby/Spread is an interface to the Spread Library at http://www.spread.org/. Spread is a toolkit and daemon that provides multicast and group communications support to applications across local and wide area networks. Spread is designed to make it easy to write groupware, networked multimedia, reliable server, and collaborative work applications." .
<_:o2>	<http://example.ruby-language.org/ rda-vocab#homepage>	"http://www.rubylinks.de/" .
<_:o2>	<http://example.ruby-language.org/ rda-vocab#download>	"http://www.rubylinks.de/download/ ruby-spread-0.1.tar.gz" .
<_:o2>	<http://www.w3.org/ 1999/02/22-rdf-syntax-ns#type>	<http://example.ruby-language.org/ rda-vocab#Product> .
<_:o1>	<http://www.w3.org/ 1999/02/22-rdf-syntax-ns#type>	<http://example.ruby-language.org/ rda-vocab#Info> .

SOAP Encoding and RDF: mapping between XML graph serialization syntaxes - There have for some years been discussions about the relationship between SOAP Encoding and RDF, as well as concerning other related representations (UML/XMI, XML Topic Maps, XLink etc.). The goal in this document has been to explore the *value* we might find in doing so, rather than treating such a mapping as a means to an end. For the SOAP developer community, value can be found through the adoption of an implementation and language neutral test cases format, and from an exploration of the wider applicability of the graph model implicit in the SOAP Encoding. By showing that SOAP Encoding is not just for RPC, some of the concerns recently aired in the 'REST' debates may be addressed. SOAP Encoding documents *can* be deployed in the Web *as documents not just message payload*. Alongside RDF/XML, XMI and other formats, they can be parsed into their abstract structure, merged with other relevant data, and queried using simple graph-matching techniques.

Conclusions: RDF and SOAP

If there is value in elaborating (through N-Triples) on the detail of the SOAP Encoding Data Model, there is probably also some value in providing background on the situation in the RDF community. RDF, like SOAP, provides a graph encoding syntax. Unlike SOAP, this syntax has a "striped" pattern. One cannot read a chunk of RDF data and read all the XML elements as labels for the edges in a graph. Consequently the RDF syntax can be hard to understand, initially. The introduction below was written to provide techniques for XML developers to quickly acquire a basic familiarity with the RDF syntax.

It should also be noted that RDF's original XML syntax was somewhat underspecfied. The 1999 RDF Model and Syntax specification did not make a clear distinction between RDF's abstract data model and its specific encoding in one form of XML. Recent work in the RDF Core WG has begun to address this. The N-Triples test case syntax has been a critical part of this work, and the utility of the approach motivates the advocacy of this same approach for SOAP Encoding. While the SOAP and RDF graph models may differ (we don't know yet), the N-Triples approach can still be a useful technique for SOAP implementors to explore.

In conclusion, this report advocates the adoption of formal, textually represented test cases for use in the Web Service community when comparing implementations of the SOAP Encoding Data Model. It further advocates the investigation of SOAP Encoded messages as a Semantic Web data format, and motivates experimentation with SOAP messages in an RDF query, storage and inference context. This work area is continued in the ☞SWAD-Europe work on 'Semantic Web Services'.

Appendix A: XSLT soap2rdf.xsl - The following XSLT transformation (by Max Froumentin) implements a

substantial but incomplete mapping from SOAP graph encoding into RDF's XML syntax.

There are a number of issues remaining with the transformation:

- in SOAP, edges have both a URI (its label) and an index (from their position in the markup). "The outbound edges of a given graph node MAY be distinguished by label or by position."
- in SOAP edges can have no target: "An edge MAY have only an originating graph node, that is be outbound only. An edge MAY have only a terminating graph node, that is be inbound only."
- we do not currently preserve datatyping information in the transformation to RDF/XML. This should be straightforward, now that RDF literal nodes can carry a datatype URIref.

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:raa="http://www.ruby-lang.org/xmlns/soap/interface/RAA/0.0.1#"
  xmlns="http://www.ruby-lang.org/xmlns/soap/interface/RAA/0.0.1#"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  version="1.0">

  <xsl:param name="body-namespace-prefix" select="'n2'"/>
  <xsl:param name="body-namespace-uri" select="'http://www.ruby-lang.org/xmlns/soap/interface/RAA/0.0.1'"/>
  <!--  <xsl:param name="schema-types-namespace-prefix" select="'xsd'"/> -->
  <xsl:param name="schema-types-namespace-uri" select="'http://www.w3.org/2001/XMLSchema'"/>


  <xsl:output indent="yes"/>

  <xsl:template match="/">
    <xsl:apply-templates/>
  </xsl:template>

  <xsl:template match="env:Envelope">
    <rdf:RDF env:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <xsl:apply-templates select="env:Body/*"/>
    </rdf:RDF>
  </xsl:template>

  <xsl:template match="env:Body/*">
    <xsl:copy>
      <xsl:apply-templates/>
    </xsl:copy>
  </xsl:template>


  <!-- now on to the meaty stuff -->

  <xsl:template match="*">
    <xsl:copy>
      <xsl:if test="@xsi:type">
        <!-- test whether this is a literal -->
        <xsl:choose>
          <xsl:when test="count(*)=0">
            <xsl:attribute name="rdf:datatype">
              <xsl:value-of select="concat($schema-types-namespace-uri,
                        '#',substring-after(@xsi:type,':'))"/>
            </xsl:attribute>
          </xsl:when>
          <xsl:otherwise>
            <xsl:if test="starts-with(@xsi:type,concat($body-namespace-prefix,':'))">
              <xsl:attribute name="rdf:parseType">Resource</xsl:attribute>

              <!-- dump the type as first child -->
              <rdf:type rdf:resource="{$body-namespace-uri}#{substring-after(@xsi:type,':')}"/>
            </xsl:if>
          </xsl:otherwise>
        </xsl:choose>
      </xsl:if>
      <xsl:apply-templates/>
    </xsl:copy>
  </xsl:template>

</xsl:stylesheet>
```
