Formalización de la semántica próxima de los lenguajes XML con UML, OWL y GRDDL

Henry S. Thompson
6 de abril de 2007

1. Resumen

Muchos lenguajes XML se definen en dos pasos: el primero, mediante la asignación de documentos XML a un modelo de datos abstracto; el segundo, mediante la definición del significado de los constituyentes del modelo de datos abstracto con respecto a un dominio. Un ejemplo obvio es (X)HTML+CSS, donde el primer paso va del documento a los cuadros anidados con propiedades, y el segundo es un conjunto de afirmaciones que los cuadros y las propiedades realizan en las representaciones. Otro ejemplo es el Esquema XML del W3C, que separa explícitamente la asignación de documentos de esquema a esquemas, por un lado, de la semántica de validación de esquemas de los componentes que conforman los esquemas, por otro.

Este artículo describe un enfoque novedoso para definir lo que se denomina la **semántica próxima** de un lenguaje XML, es decir, la correspondencia entre los conjuntos de información XML y los modelos de datos (abstractos) específicos del lenguaje. El enfoque consta de tres partes:

- Un conjunto de convenciones para construir modelos UML, utilizando el editor de diagramas UML gráficos de código abierto Violet;
- Una tubería de hojas de estilo XSLT para convertir la representación XML de esos diagramas en ontologías OWL;
- Un conjunto de pautas para escribir hojas de estilo XSLT u otras transformaciones (por ejemplo, canalizaciones) para implementar el mapeo activado por GRDDL desde documentos de lenguaje a instancias de modelos de datos expresados en RDF.

El resultado de implementar este enfoque es que una ontología OWL para un modelo de datos de lenguaje y una instancia RDF correspondiente a un documento de lenguaje individual pueden combinarse y comprobarse su consistencia. Si el resultado es consistente, también puede compararse con expresiones RDF de instancias de modelos de datos concretos de una implementación. Esto permitiría realizar pruebas de conformidad semiautomáticas si la especificación del lenguaje incluyera las tres partes mencionadas anteriormente.

A lo largo del artículo, los puntos en discusión se ilustran con ejemplos del lenguaje del Modelo de Procesamiento XML, actualmente en desarrollo por el Grupo de Trabajo del Modelo de Procesamiento XML del W3C. Se aprendieron algunos aspectos sobre dicho lenguaje durante el ejercicio que se describe aquí, los cuales también se discuten brevemente.

También se establecen conexiones con trabajos anteriores sobre la expresión de información de enlace de datos a través de anotaciones de esquema, que sugieren la posibilidad de generar automáticamente las hojas de estilo necesarias para la parte (3) anterior en algunos casos.

2. Agradecimientos

Dan Connolly escribió la primera hoja de estilo de Violet a OWL, que realmente inspiró todo este proyecto.

3. Definiciones/implementaciones de lenguajes (XML) en dos partes y tres niveles

Todos los lenguajes formalmente definidos, salvo los más simples, se comprenden comúnmente y suelen definirse sin pasar directamente de la notación al significado. Se recurre a un nivel intermedio, más o menos explícito, a veces denominado «sintaxis abstracta», «forma subyacente» o «modelo abstracto de datos». Las implementaciones concretas de estos lenguajes suelen materializar de forma más o menos directa este nivel intermedio en las estructuras de datos del sustrato de implementación.

Por lo tanto, tanto las especificaciones como las implementaciones suelen constar de dos partes: la asignación de la notación al nivel intermedio y la importación (en teoría o en la práctica) de la composición de dicho nivel. Este artículo se centra en la primera parte, principalmente desde la perspectiva de los productores y consumidores de especificaciones.

La tesis de este artículo es que las especificaciones pueden y deben incluir definiciones operativas y completamente explícitas tanto del nivel intermedio como del mapeo. Se propone un conjunto específico de tecnologías para lograr esto cuando el lenguaje en cuestión utiliza XML para su notación.

La tarea naturalmente se divide en dos partes: definir el nivel medio (es decir, el modelo de datos abstracto) y mapear instancias individuales de XML a su forma de nivel medio.

4. El nivel medio

La primera parte de nuestro proyecto requiere que produzcamos un modelo formal y explícito del nivel medio.

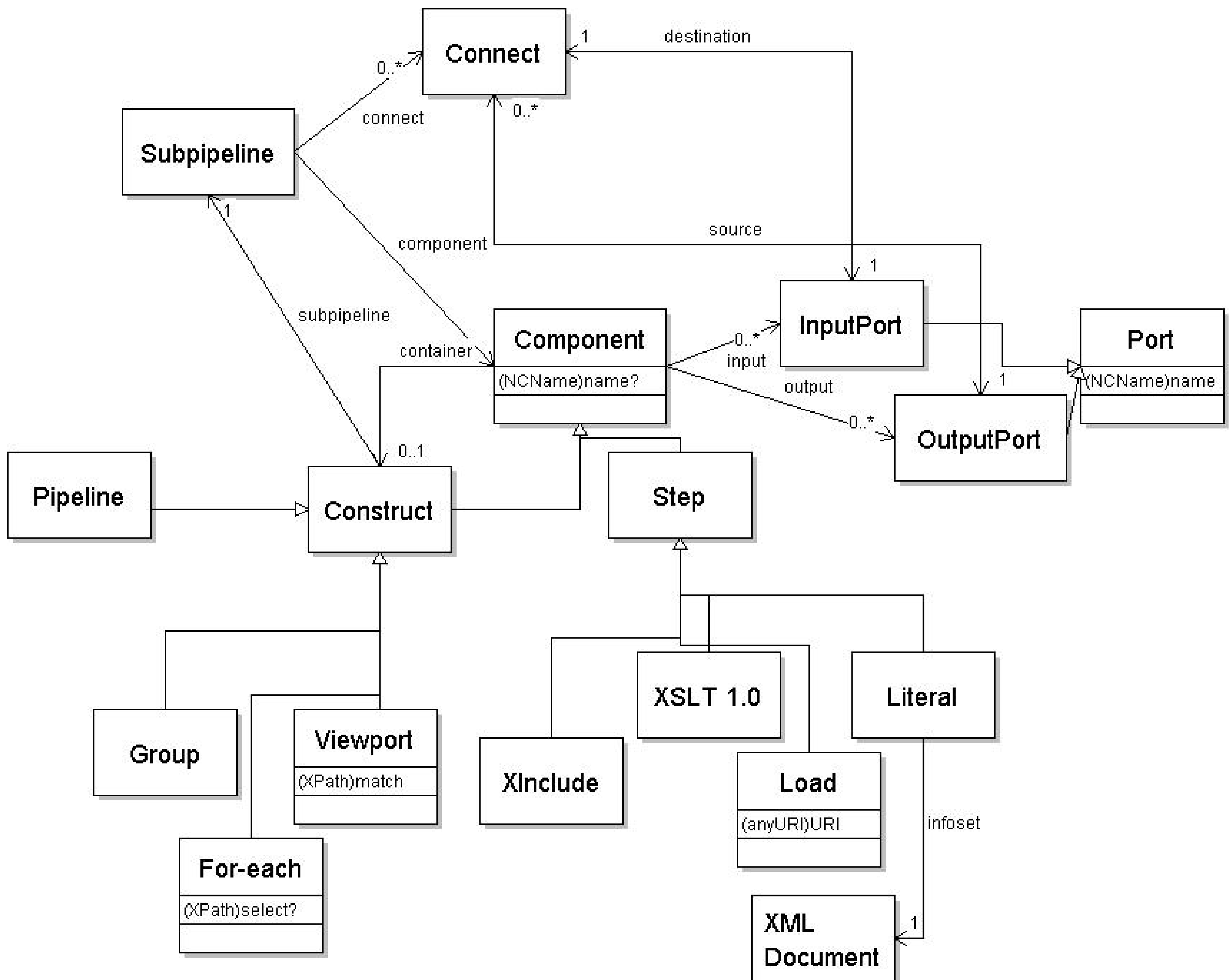
4.1. Definición del nivel medio

Nada supera a los lápices de colores y una pizarra para diseñar modelos de objetos, que es precisamente de lo que se trata. Sin embargo, el paso a XML carece de soporte técnico, así que la mejor alternativa es un programa de dibujo. No uno de uso general, sino uno diseñado para dibujar modelos de objetos. Y que genere XML. Y debería ser gratuito.

UML es un candidato obvio: está diseñado para uso gráfico y existen herramientas gratuitas para dibujar diagramas UML sencillos fácilmente. La que encontré y he utilizado para el trabajo que se reporta aquí es **Violet** [Horst05]. Es importante destacar que en este ejercicio *no* estoy usando UML. En su lugar, estoy usando una herramienta de dibujo UML y un subconjunto limitado de las convenciones UML para dibujar modelos de objetos:

1. La jerarquía de clases (con herencia implícita);
2. La distinción entre propiedad y atributo;
3. Anotaciones de cardinalidad y direccionalidad de propiedades.

Cada uno de ellos se ilustra en el siguiente diagrama para un subconjunto muy simplificado de XProc [Wal07], el lenguaje de canalización XML que actualmente está desarrollando el Grupo de trabajo del modelo de procesamiento XML del W3C:



- Cada cuadro es una clase, y las flechas que terminan en triángulos abiertos indican relaciones entre subclases y superclases. Por ejemplo, hay dos tipos de componentes: pasos y construcciones, y cuatro tipos de construcciones: tuberías, grupos, para-cada uno y ventanas gráficas.
- Los atributos se dan dentro de cuadros de clase, con los tipos entre paréntesis y la opcionalidad indicada con un '?' final.
- Properties are notated with simple arrows. Cardinalities are given at the *far* end, so for example a Subpipeline has a **connect** property, which may be present zero or more times, and whose value is a Connect.
- A double-headed arrow indicates a bi-directional property. For example a Connect has a single **destination**, which is an Input Port, and this relation is 1-to-1 and can be reversed, that is, an Input Port has a **destination** port as well, whose value is the Connect.

Violet uses Java's XML serialiser to dump its pictures, and the next step is accordingly to convert this format to something more congenial. I use a pipeline (!) of three XSLT steps to transform the Violet output into an OWL [OWL04] ontology. The figure below illustrates (selectively) how various aspects of the object model are translated:

```
     <rdf:RDF xml:base="http://www.w3.org/2007/03/xproc">

[1]   <owl:Class rdf:ID="Component">
       <rdfs:subClassOf>
[5]     <owl:Restriction>
         <owl:onProperty rdf:resource="#name"/>
[7]      <owl:maxCardinality
              rdf:datatype="&xsd;#int">1</owl:cardinality>
        </owl:Restriction>
       </rdfs:subClassOf>
       <rdfs:subClassOf>
        <owl:Restriction>
         <owl:onProperty rdf:resource="#container"/>
[7]      <owl:maxCardinality
             rdf:datatype="&xsd;#integer">1</owl:maxCardinality>
        </owl:Restriction>
       </rdfs:subClassOf>
      </owl:Class>

[2]   <owl:DatatypeProperty rdf:ID="name">
       <rdfs:domain> rdf:resource="#Component"/>
       <rdfs:range rdf:resource="&xsd;.xsd#NCName"/>
      </owl:DatatypeProperty>

[3]   <owl:ObjectProperty rdf:ID="input">
       <rdfs:domain> rdf:resource="#Component"/>
       <rdfs:range rdf:resource="#InputPort"/>
      </owl:ObjectProperty>

[3]   <owl:ObjectProperty rdf:ID="container">
       <rdfs:domain> rdf:resource="#Component"/>
       <rdfs:range rdf:resource="#Construct"/>
      </owl:ObjectProperty>

[1]   <owl:Class rdf:ID="Construct">
[4]   <rdfs:subClassOf rdf:resource="#Component"/>
       <rdfs:subClassOf>
        <owl:Restriction>
         <owl:onProperty rdf:resource="#subpipeline"/>
[6]      <owl:cardinality
              rdf:datatype="&xsd;#int">1</owl:cardinality>
        </owl:Restriction>
       </rdfs:subClassOf>
      </owl:Class>

      <owl:ObjectProperty rdf:ID="subpipeline">
       <rdfs:domain rdf:resource="#Construct"/>
       <rdfs:range rdf:resource="#Subpipeline"/>
      </owl:ObjectProperty>

     </rdf:RDF>
```

All the key aspects of the OWL representation of the data model, as translated automatically from the quasi-UML diagram, are illustrated in this extract (the items below are keyed by number to the figure above):

1. Classes are . . . `owl:Classes`
2. Attributes are `owl:DatatypePropertys`
3. Properties are `owl:ObjectPropertys`
4. The class hierarchy is drawn using `rdfs:subClassOf`
5. Cardinality constraints are handled using subclassing and `owl:Restriction`
6. Required attrs/props have `owl:cardinality` restricted to `1` (see section ??? below for some discussion of this)
7. Optional attrs/props have `owl:maxCardinality` restricted to `1`

Note that no cardinality constraints are present for e.g. the `input` property, as the UML range `0..*` in the diagram means there *is* no constraint.

4.2. Working with the middle level

There are a number of free tools available for browsing and, to a limited extent, checking OWL ontologies. I've found SWOOP [SWOOP06], Protégé [Prot07] and Pellet [Pellet06] helpful in various ways (see also section ??? below). I also found it helpful to write a simplified version of the middle-level construction pipeline to simply extract and display all the properties from a Violet UML diagram, for example:

Component -- output 0..* --> OutputPort

Component	--	input	0..*	-->	InputPort
Component	--	container	0..1	-->	Construct
Connect	<-- 1	destination	1	-->	InputPort
Connect	<-- 0..*	source	1	-->	OutputPort
Construct	--	subpipeline	1	-->	Subpipeline
Literal	--	infoset	1	-->	XML Document
Subpipeline	--	component	1..*	-->	Component
Subpipeline	--	connect	0..*	-->	Connect

Developing the UML and inspecting and checking the corresponding OWL ontology interacted positively with the design process of the XProc language itself. When I couldn't determine exactly what to put in the model at certain points, I could sometimes identify under-specification in the language definition. I could then feed questions back to the Working Group to get the uncertainty resolved in a subsequent draft. I also was able to detect several places in which terminology was used in inconsistent or conflicting ways in different parts of the specification.

On the other hand, a simple class hierarchy with attributes and properties is a pretty modest modelling tool, and it is not surprising that not every difficulty in modelling represents a problem with the language being modelled. In particular, it is easy to state exceptions in the prose of a specification, but not always easy to capture them in a model. For example, the model, even in its simplified form, suggests that the 'container' property of **Component**s is optional. If fact, it is obligatory for all components *except* the top-level pipeline. It would have been possible to express this in the model, but the resulting multiplication of classes would have been a very high price to pay in terms of perceived complexity.

The complete model in both Violet/UML and OWL, as well as the stylesheets needed to produce one from the other, are available online:

UML
: http://www.w3.org/XML/XProc/2007/04/modelling/xproc.violet, http://www.w3.org/XML/XProc/2007/04/modelling/xprocOntology.png

OWL
: http://www.w3.org/XML/XProc/2007/04/modelling/xproc.owl

Property display
: http://www.w3.org/XML/XProc/2007/04/modelling/xprocProps.html

Violet to OWL
: http://www.w3.org/XML/XProc/2007/04/modelling/violetToRDF1.xsl, http://www.w3.org/XML/XProc/2007/04/modelling/violetToRDF2.xsl

Violet to property display
: http://www.w3.org/XML/XProc/2007/04/modelling/violetToRDF1.xsl, http://www.w3.org/XML/XProc/2007/04/modelling/violetProperties.xsl

5. Individual instances

The second part of our project requires us to integrate individual instances (documents in the language at hand) with the abstract data model, that is, to map from the XML notation of the language into appropriately decorated and connected instances of the classes in that model. Conformance to the language definition can be checked during the mapping process and at the end.

5.1. Mapping from XML (using GRDDL) -- Some design principles

The idea that the (in our case proximate) semantic content of an XML document can be mechanically derived from the document itself is the core proposition of the **Gleaning Resource Descriptions from Dialects of Languages** work [GRDDL07]. GRDDL provides conventions, both at the individual document level and at the level of namespaces or languages, for associating information about how to map from XML documents to their semantic content as notated using RDF. Although GRDDL singles out XSLT as the obvious means for expressing such mappings, it also mentions that pipelines of XSLT and other steps may be appropriate in complex cases, and we will indeed suggest that multiple steps are often appropriate.

"Divide and conquer" is an excellent software engineering design principle to use when confronted with what is, essentially, a compilation task. Breaking the overall transformation down into smaller steps not only facilitates debugging and maintenance, but also can shorten development time by facilitating understanding. In the particular case of the XProc language, the pipeline which effects the mapping, discussed briefly below, consists of six steps, all currently expressed using XSLT.

Two examples will suffice to illustrate the benefits of decomposing the mapping into stages:

- The very first step in the pipeline operates solely on elements with attributes whose value is a QName. All it does is add two *additional* attributes in every such case, one with the local name and the other the namespace name, if any, resulting from looking up the appropriate namespace binding. This enables all the subsequent steps to operate on that pair of attributes, instead of having to reproduce the moderately complex testing, string surgery and lookup code whenever it was needed. This considerably improves readability.

- XProc provides for quite a lot of defaulting of the connections between steps, by appeal to an *environment* which is recursively specified. The mapping process approaches this by constructing the environment in one step, and the connections in a subsequent step, appealing to the environment as necessary.

Another design principle which proved very useful was to organise the main mapping from the XML notation of pipeline steps into their RDF equivalents in terms of the class hierarchy itself. Named templates were defined for all but the lowest level of the class hierarchy, to carry out those aspects of the mapping which were appropriate to that level. Looking back at Figure ???, what this means is processing e.g. an `p:for-each` would involve not only a template which matches that element alone, but also a call to a named template for **Construct**s, and then in turn a call to a named template for **Components**.

The first, middle and last templates in this chain (and this would generalise to greater depth as needed) have a regular (and similar) structure:

```
<xsl:template match="[some element name]">
 <correspondingClassName>
  [add attributes as appropriate to this level alone]
  <xsl:call-template name="[superClass]"/>
  [add children as appropriate to this level alone]
 </correspondingClassName>
</xsl:template>

<xsl:template name="[intermediate Class]">
 [add attributes as appropriate to this level alone]
 <xsl:call-template name="[superClass]"/>
 [add children as appropriate to this level alone]
</xsl:template>

<xsl:template name="[top Class]">
 [add attributes as appropriate to top level]
 [add children as appropriate to top level]
</xsl:template>
```

So for instance in the `p:for-each` case, this would play out ore or less as follows:

```
<xsl:template match="p:for-each">
 <For-each>
  <xsl:call-template name="Construct"/>
  <xsl:if test="@select"><select><xsl:value-of select="@select"/></select></xsl:if>
 </For-each>
</xsl:template>

<xsl:template name="Construct">
 <xsl:call-template name="Component"/>
 <subpipeline>
  <Subpipeline>...</Subpipeline>
 </subpipeline>
</xsl:template>

<xsl:template name="Component">
 <xsl:attribute name="rdf:about">...<xsl:value-of select="@name"/>...</xsl:attribute>
 . . .
</xsl:template>
```

The standard pattern is reflected here. The element in the document itself determines the RDF class to create the instance, the **Construct** class template handles the **subpipeline** relation which is common to all constructs, and the **Component** class template gives the instance its RDF identity.

5.2. Checking

The mapping process provides the opportunity to check language constraints both using XSLT at the syntactic level for constraints which may not have been captured in a schema, and using ontology processors and/or SPARQL on the resulting RDF instance.

5.2.1. Checking the syntax with XSLT

For example, the XProc specification says: "It is a static error if the port specified by a `p:pipe` is not in the readable ports of the environment." This can be checked by the part of the mapping pipeline, mentioned above, which connects ports based on the contents of the environment, as computed by previous stages of the mapping.

5.2.2. Checking the RDF

Once a document has been mapped into RDF, the result, along with the ontology for the model whose construction was discussed in section ??? above, can be loaded into an ontology tool or 'reasoner' for consistency checking. For example, the Pellet 'reasoner' can detect maxCardinality violations (if it is instructed to treat all named individuals as distinct). This will detect multiple bindings to the same input port (if this has not been caught by schema or during mapping). During development, such a tool may also detect errors in the mapping itself.

The most common cardinality problem, particularly during model development, but also occurring with instances, is minCardinality failure, that is, the absence of some required property or attribute. It is in the nature of RDF/OWL that a 'reasoner' does not detect such errors, as they don't result in a contradiction -- one should understand a minCardinality assertion as saying "There is one, even if we haven't identified it yet." We can, however, detect missing information using a SPARQL [SPARQL06] query, as follows:

```
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
 SELECT ?x ?p
 WHERE {
  ?r a owl:Restriction .
  { ?r owl:cardinality 1 } UNION { ?r owl:minCardinality 1 } .
  ?r owl:onProperty ?p .
  ?p rdfs:domain ?c .
  ?x a ?c .
  OPTIONAL {?x ?p ?y}
  FILTER (!bound(?y)) }
```

This can be read as saying "Find any relevant cardinality restrictions, then check all the instances of the domains of the restricted properties to see that they do indeed have a real value for that property."

5.3. Mapping stylesheets and example

The stylesheets for mapping from instances to models, along with an example based on Figure 1 of the XProc draft, are available:

XProc XML to RDF XML
: http://www.w3.org/XML/XProc/2007/04/modelling/pipeAS0.xsl, http://www.w3.org/XML/XProc/2007/04/modelling/pipeAS1.xsl, http://www.w3.org/XML/XProc/2007/04/modelling/pipeAS2.xsl, http://www.w3.org/XML/XProc/2007/04/modelling/pipeAS3.xsl, http://www.w3.org/XML/XProc/2007/04/modelling/pipeAS4.xsl

Example XML and RDF
: http://www.w3.org/XML/XProc/2007/04/modelling/fig1.xml, http://www.w3.org/XML/XProc/2007/04/modelling/fig1.owl

6. Conclusions

Whether the availability of an easily inspected quasi-reference implementation of the first phase of implementing the XProc pipeline language will benefit implementers and users in the long run is impossible to say at this point. It is often remarked that early implementation aids specification, and that has certainly happened in this project. It has generated significant amounts of feedback into the specification process, in terms of questions, suggestions and observations.

Perhaps the most significant observation about the language which emerged from the model development work was that by far the easiest way to handle the duality of input and output ports on constructs such as **Viewport** (and **Pipeline** itself), was to recognise that despite their surface syntax, the things notated as `p:input` and `p:output` children of e.g. `p:viewport` were in fact instances of *both* **InputPort** *and* **OutputPort**.

Another potentially useful side-effect of the modelling effort was that it became necessary to introduce a new **PipelineShell** class, to hold the top-level pipeline and the contents of any imported libraries.

Finally, the very fact that a pipeline of stylesheets was by far the easiest way to specify the mappings involved is itself an encouraging comment on the importance of the XProc language. Although that *particular* benefit is unlikely to be relevant for efforts to specify other XML languages, I hope the other benefits of the particular instance of the "It's *all* just XML" philosophy illustrated by the methodology described here will be of just such wider relevance.

7. Bibliography

GRDDL07
: Connolly, D. ed. *Gleaning Resource Descriptions from Dialects of Languages (GRDDL)*, W3C, 2007. Available online as http://www.w3.org/TR/grddl/.

Pellet06
: Sirin, E., B. Parsia et al. "Pellet: A practical OWL-DL reasoner", *Journal of Web Semantics* (To Appear), 2006. Available online at http://www.mindswap.org/papers/PelletJWS.pdf. Software download, documentation, etc. available online at http://pellet.owldl.com/.

Horst05
: Horstmann, C *The* ***Violet*** *UML diagram editor*, 2005. Documentation, demo app and download available online at http://www.horstmann.com/violet/

SWOOP06
: Kalyanpur, A., B. Parsia, et al. "Swoop: Un navegador de edición de ontologías web", *Journal of Web Semantics* , v. 4, n.º 2, Science Direct, 2006. Disponible en línea en http://www.websemanticsjournal.org/ps/pub/2006-7 . Descarga de software, etc., disponible en http://code.google.com/p/swoop/ .

BÚHO04
McGuinness, D y F. van Harmelen, eds. *OWL Web Ontology Language Overview* , W3C, 2004. Disponible en línea como http://www.w3.org/TR/owl-features/ .

Prot07
*El editor de ontologías **Protégé*** , Universidad de Stanford, 2007. Descarga de software, documentación, etc. disponibles en línea en http://protege.stanford.edu/ .

SPARQL06
Prud'hommeaux, E. y A. Seaborne eds. *Lenguaje de consulta SPARQL para RDF* , W3C, 2006. http://www.w3.org/TR/rdf-sparql-query/ .

Wal07
Walsh N. y A. Milowski, eds. *XProc: Un lenguaje de canalización XML* , W3C, 2007. Disponible en línea en http://www.w3.org/TR/xproc/