Preserving modularity in XML encoding of description logics

Jérôme Euzenat
INRIA Rhône-Alpes
655 avenue de l'Europe, 38330 Montbonnot, France
Jerome.Euzenat@inrialpes.fr

Abstract

Description logics have been designed and studied in a modular way. This has allowed a methodic approach to complexity evaluation. We present a way to preserve this modularity in encoding description logics in XML and show how it can be used for building modular transformations and assembling them easily.

1 Motivation

In the words of Tim Berners-Lee, the "semantic web" [2] requires a set of languages of increasing expressiveness such that anyone can pick up the right language for each particular semantic web application. This is the approach taken by the description logic community over the years.

We show how the modularity of description logics, that has mainly be used at a theoretical level, can be turned into an engineering advantage in the context of the semantic web. For that purpose, we introduce a description logic markup language (DLML) which encodes description logics in XML and preserves their modularity. We then present transformations that are described on individual constructors and can be composed to form more complex transformations. By factoring out syntactic rules and processing methods, the same set of transformations can apply to many logics.

The next section presents the syntactic encoding of description logics in XML. The third section illustrates the use of transformation of the XML form in order to achieve relatively complex transformations.

2 Modular encoding

Description logics are introduced first, before XML. XML is presented through a first application to description logics. Then, the advantage of preserving the modularity of description logics in the encoding is discussed and the corresponding set of Document Type Descriptions (DTD) is presented.

2.1 Description logics

Description logics manipulate terms. Below are two (concept) term descriptions stating that a `modernfirm` is a `firm` in which all `operational` employees are `bachelor`.

$$\lceil \texttt{modernfirm} \doteq \texttt{firm} \sqcap (\forall \texttt{operational}.\texttt{bachelor})$$
$$\texttt{startup} \doteq \texttt{firm} \sqcap (\forall \texttt{operational}.\texttt{phD}) \rfloor$$

Two kinds of terms are involved: concepts (such as `firm`) and roles (such as `operational`). Their descriptions are built from sets C (resp. R) of atomic concept (resp. role) names and term constructors. Here, the main logic considered is $\mathcal{ALC}$ [5]. $\mathcal{ALC}$ contains special atomic terms $\top$ and $\perp$ and makes only use of concept constructors for conjoining or intersecting two concepts c and c' ($c \sqcap c'$ or $c \sqcup c'$), restricting a the codomain of a role r to a concept c ($\forall r.c$), taking the complement of a concept c ($\neg c$) or asserting the existence of a role ($\exists r$).

Terms have a set-theoretic model semantics. An interpretation $\mathcal{I}$ over a domain $\mathcal{D}$ is a mapping $\mathcal{I} : C \rightarrow 2^{\mathcal{D}}$ and $R \rightarrow 2^{\mathcal{D}\times\mathcal{D}}$ such that:

$$\mathcal{I}(\top) = \mathcal{D} \qquad \mathcal{I}(\perp) = \emptyset \qquad \mathcal{I}(\neg c) = \mathcal{D} - \mathcal{I}(c)$$
$$\mathcal{I}(c \sqcap c') = \mathcal{I}(c) \cap \mathcal{I}(c') \qquad \mathcal{I}(c \sqcup c') = \mathcal{I}(c) \cup \mathcal{I}(c')$$
$$\mathcal{I}(\forall r.c) = \{o \in \mathcal{D} | \forall y \in \mathcal{D}, \langle o, y\rangle \in \mathcal{I}(r) \Rightarrow y \in \mathcal{I}(c)\}$$
$$\mathcal{I}(\exists r) = \{o \in \mathcal{D} | \exists y \in \mathcal{D}, \langle o, y\rangle \in \mathcal{I}(r)\}$$

A terminology is a set of assertions of type $a \doteq c$ or $a \dot{\leq} c$. They are interpreted as definitions or descriptions of atomic terms (a). An interpretation $\mathcal{I}$ is said to satisfy an assertion α (noted $\models_{\mathcal{I}} \alpha$) if:

$$\models_{\mathcal{I}} a \doteq c \text{ iff } \mathcal{I}(a) = \mathcal{I}(c),$$
$$\models_{\mathcal{I}} a \dot{\leq} c \text{ iff } \mathcal{I}(a) \subseteq \mathcal{I}(c)$$

As usual, a model of a terminology is an interpretation $\mathcal{I}$ that satisfies all the assertions of the terminology.

2.2 XML

XML [4] is a markup language recommended by the "Worldwide web consortium" (W3C). It aims at being a document exchange format between applications. It is voluntarily a trade-off between the simplicity of HTML and the power of SGML. Unlike HTML and like SGML, XML is extensible through a Document Type Description (DTD).

The syntax of XML documents is very simple and can be summarized by: `<TAG att1=v1 ... attn=vn/>` or `<TAG att1=v1 ... attn=vn>content</TAG>` where `content` is a sequence of XML expressions and unparsed character strings, `atti` is an attribute identifier and `vi` the value of the corresponding attribute (a string). The `TAG` is called element. An XML document is seen as a tree whose root is the document and in which the children of a node are whatever in its content.

Here is the beginning of an XML document introducing the definition of concept `modernfirm`:

```
<?xml version="1.0"?>
<!DOCTYPE TERMINOLOGY SYSTEM "simple-alc.dtd">

<TERMINOLOGY>...
  <CDEF>
    <CATOM NAME="modernfirm"/>
    <AND>
      <CATOM NAME="firm"/>
      <ALL>
        <RATOM NAME="operational"/>
        <CATOM NAME="bachelor"/>
      </ALL>
    </AND>
  <CDEF>
  ...
</TERMINOLOGY>
```

The knowledge base (TERMINOLOGY) contains a definition (CDEF) equating the concept (CATOM) `modernfirm` to the term made of the conjunction (AND) of the concept (CATOM) `firm` with the restriction (ALL) of the role (RATOM) `operational` to the concept (CATOM) `bachelor`. It is exactly the same definition as above. This document is governed by a DTD (!DOCTYPE). A DTD describes each element by specifying:

- Its content as a regular expression using the sequence (,) and alternative (|) in function of a variable number (1/?/+/*) of other elements ;

- Its attributes by specifying the kind of value taken (and some other parameters).

The above example can be defined by the following DTD fragment:

```
<!ELEMENT TERMINOLOGY (CDEF|CPRIM)*>

<!ELEMENT CDEF (CATOM,(CATOM|AND|ALL|...))>

<!ELEMENT AND ((CATOM|AND|ALL|...)*)>
<!ELEMENT ALL (RATOM,(CATOM|AND|ALL|...))>
 ...

<!ELEMENT CATOM EMPTY>
```

```
<!ATTLIST CATOM NAME    CDATA    #REQUIRED>
<!ELEMENT RATOM EMPTY>
<!ATTLIST RATOM NAME    CDATA    #REQUIRED>
```

It defines the elements visible in the document: (TERMINOLOGY, CDEF, CATOM, AND ...). Each one is defined by its (possibly empty) content (keyword !ELEMENT) and its attributes (optional keyword !ATTLIST). A third keyword !ENTITY, is used for introducing text blocks to be reused literally. The content is specified with regard to other elements and character strings. The attributes can be more complex; here they are typed by character strings (CDATA) and required (#REQUIRED).

An XML document is said well-formed if it complies with the XML syntax (i.e. if the interleaving of opening and closing tags is a well-parenthesized expression). It is said valid if it satisfies the constraints expressed in its DTD. The above example is valid with regards to its DTD.

2.3 Modular encoding of description logics

If the DTD above is usable, it has the drawback of containing the constructor specifications in its body. Modularity is an important issue for description logics since various description logics are defined by adding constructors to other ones. It is thus useful, when designing a system of DTD for description logics to preserve that property of defining independently concept and role constructors and term definers and assembling them into a particular logic.

The modular encoding of the description logics is made of three kind of DTD: atoms (introducing the atomic terms), operators (e.g. $\forall$, $\sqcap$, $\neg$) and formula constructors (e.g. $\dot{\leq}$, $\doteq$). An arbitrary number of these files are put together in order to form a particular logic.

For instance below is the content of the DTD of the INV (converse of a role) constructor:

```
<!ELEMENT INV (%RDESC;)>
```

It refers to the RDESC entity which is a placeholder for all the possible role constructors. This placeholder is not defined in the operator DTD, but is set, for each individual logic, to the available role constructors.

We have also defined the notion of Document Semantic Description (DSD) which enables the description the formal semantics of an XML language (just like the DTD or schemas expresses the syntax). The DSD language defined in XML takes advantage of Xpath for expressing references to sub-expressions and MathML for expressing the mathematical gear. To the DTD of INV is associated a DSD describing the semantics of the operator (i.e. $I((\texttt{inv r})) = I(\texttt{r})^{-1}$):

```
<dsd:DSD>
<dsd:denotation match="dl:INV">
```

```
    <mml:eq/>
       <mml:apply>
          <mml:inverse/> <!-- converse for binary rels -->
          <dsd:apply-interpretation select="*[1]"/>
       </mml:apply>
</dsd:denotation>
</dsd:DSD>
```

DLML provides the DTD and DSD of all the covered operators and is able to build automatically from the description of a logic those of that logic. This is achieved through a DLML logic description file which is described as follows:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE dlml:logic SYSTEM "dlml.dtd">

<dlml:logic version="1.0" name="alc">
  <dlml:atoms/>

  <dlml:cop name="anything"/>
  <dlml:cop name="nothing"/>
  <dlml:cop name="and"/>
  <dlml:cop name="or"/>
  <dlml:cop name="anot"/>
  <dlml:cop name="not"/>
  <dlml:cop name="all"/>
  <dlml:cop name="some"/>

  <dlml:cint name="cdef"/>
  <dlml:cint name="cprim"/>
</dlml:logic>
```

From this description, two XSLT stylesheets are able to generate the DTD and DSD corresponding to the language. They can be used for expressing $\mathcal{ALC}$ terminologies in XML like the one above.

Every constructor is thus defined only once and has just to be mentioned for describing a new logic. The specification of a particular logic is achieved by declaring the set of possible constructors and the logic's DTD is automatically build up by just assembling those of elementary constructors. The actual system contains the description of more than 40 constructors and 25 logics.

DLML has been built around DTD because there was not much support for other schema languages in XML parsers. The XML Schema language [11] would ease the implementation of DLML by replacing the use of the RDESC macro for a class hierarchy (all the operators being defined in function of abstract classes `concept` and `role` whose possible instanciation would be determined by the current logic).

3 Transformations

What can such a DTD for description logics be good for? Once a language is encoded in XML it is very convenient to use XSLT [6] in order to transform syntactically a representation into another one. The first application is the import and export of terminologies from a description logic system. We have developed several transformation stylesheets for importing and exporting ontologies from FACT [1], OIL [7], DAML-ONT [10] and syllogistic representation. We are concerned here with the transformations that are tied to the DLML encoding and which takes advantage of this modularity.

3.1 Modular transformations

The first kind of transformation is the transformation from XML to layout. This is exactly what has been implemented providing the layout of the terminology given above (between $\lceil$ and $\rfloor$). The LaTeX has been generated automatically from the XML document describing the terminology through an XSLT stylesheet. Moreover, the description of the transformation is given in the same modular way as above. Below is the template corresponding to the translation of the ALL constructor:

```
<xsl:template match="dl:ALL">
  <xsl:text>\forall </xsl:text>
  <xsl:apply-templates  mode="set-role" select="*[1]"/>
  <xsl:text>.</xsl:text>
  <xsl:apply-templates  mode="set-concept" select="*[2]"/>
</xsl:template>
```

It only tells that when it has to transform a document fragment tagged by ALL, it generates the LaTeX for $\forall$ (`\forall`) applies the transformation to its role argument generates a dot (.) and then applies the transformation to its second argument.

Transformation from a knowledge representation formalism to another is the most important application. Considering the description logics expressed in DLML as a family of languages has the advantage that many such transformation are simple to define because constructors can be shared and they have the same semantics. The next section will present some of these transformations.

Normalization (used in “normalize and compare” subsumption test strategies [3]) can also be attempted through XML transformations. However, normalization is difficult to implement with XSLT which attempts to prohibits (non structure-based) recursive operations (and thus closure). However, this could be implemented through another language.

3.2 Proof-based transformations

In some restricted context, transformations can be forged directly from proofs. In the DLML context, the languages have the same syntactic structure and the semantics of the operators remains the same across languages so it is easier to do it. The $\mathcal{ALC}$ and $\mathcal{ALUE}$ languages are known to be equivalent. The proof of equivalence is a demonstration that any operator missing in one language can be expressed in the other language (preserving interpretations). This iterative proof can be expressed, by a human being, as:

$$\forall \langle D, I \rangle, \ldots$$
$$\begin{aligned} I((\texttt{not Nothing}))) &\equiv I(\texttt{Anything}) \\ I((\texttt{not } c)) &\equiv I((\texttt{anot } c)) \qquad\qquad \text{for } c \in \mathcal{N}_C \\ I((\texttt{not } (\texttt{anot } c))) &\equiv I(c) \\ I((\texttt{not } (\texttt{all } r\ c))) &\equiv I((\texttt{csome } r\ (\texttt{not } c))) \ldots \end{aligned}$$

It is straightforward to transform that proof into the following XSLT templates:

```
 ...
<xsl:template mode="process-not" match="dl:NOTHING">
  <dl:ANYTHING/>
</xsl:template>

<xsl:template mode="process-not" match="dl:CATOM">
  <dl:ANOT><xsl:apply-templates select="."/></dl:ANOT>
</xsl:template>

<xsl:template mode="process-not" match="dl:ANOT">
  <xsl:apply-templates select="*"/>
</xsl:template>

<xsl:template mode="process-not" match="dl:ALL">
  <dl:CSOME>
    <xsl:apply-templates select="*[1]"/>
    <xsl:apply-templates mode="process-not" select="*[2]"/>
  </dl:CSOME>
</xsl:template>
 ...
```

The last rule tells that when encountering a `ALL` in the scope of a negation (`mode="process-not"`), it is replaced by a `CSOME` with the non-negated transformation of the first argument and the negated transformation of the second one.

This can be made even more modular. With Heiner Stuckenschmidt, we have developed a transformation flow for importing DAML-ONT and OIL ontologies in the $\mathcal{SHIQ}$ logic. In addition to importing the formalisms in DLML, this flow composes four transformations:

- a merging transformation that combine both ontologies (by adding the two terminologies syntactically);

- a transformation that convert the expression using `DOMAIN` into expression using `ALL` and `INV`;

- a transformation that converts the expressions using ONEOF and INDIVIDUAL into expressions using OR, CATOM and CEXCL ($\dot{\oplus}$);
- a transformation that converts the expressions using CEXCL into expressions using CPRIM and NOT.

It can be proved that each of these unit transformations preserve the consequences of the initial terminology (either through model preservation or model homomorphism). Moreover, these transformations, since they are not particularly tied to some logic, can be reused in other transformation flows.

3.3 Toward proof-checking of transformations

DSD can be used for a variety of purposes:

documenting language semantics for the user or the application developer who will require a precise knowledge of the semantics of constructs. This is eased by a transformation from DSD to LaTeX.

computing interpretations from the input of the base assignment of the variables.

checking proof of transformations is a very promising application in the line of the "web of trust" idea [2].

proving transformations in an assisted or automatic way;

inferring transformations from the semantics description is a very hard problem. However, from a given proof, it can be easier.

One of the more promising use is proof-checking. We said above that transformations could be build from proofs. It is useful to be able to represent these proofs. So, the equivalence between two logics can be established by proving that a transformation from one logic to another preserves the models (in the sense of model theory). Having the description of the semantics of both languages and the proof of a transformation enables the application of the proof-carrying code framework [8] to the importation of knowledge written in a language into another. This ensures that the resulting knowledge is equivalent to the initial representation (or satisfies the proved properties).

This approach leads directly to a framework in which transformations from one representation language to another are available from the network and proofs of various properties of these languages are attached to them. It is noteworthy that transformations and proofs do not have to come from the same origin.

The transformation system engineer can gather these transformations and their proofs, check the proofs before importing them in its transformation development environment. She will then be able to create a new transformation flow and generate the proofs of the properties that she requires. Finally, she will be able to release the transformation and its proof on the network.

4 Related work

Encoding description logics in XML is not a very original neither a very difficult task. What is slightly more original is the encoding of description logics that preserve their modularity in a useful way.

There has been some interest in the description logics community about interoperability through XML. In particular the FACT system has offered some encoding of its language into XML [1]. The OIL language [7] which stemmed from that effort is expressed in XML and RDF (we have used the XML version for developing transformations, but as soon as RDF is encoded in XML as in DAML-ONT, it is not problematic to transform them). The FACT and OIL DTD are similar to the kind of DTD developed in §2.2. However these DTD aim at exchanging knowledge between several FACT or OIL reasoners instead of various systems as advocated here. As a consequence, the modularity and semantic description issues discussed here have not been taken into account. Of course, one can refrain from using some constructors, thus using a sub-logic in another one but no DTD exists for the sub-language.

The KRSS specification [9] has been created for being the interlingua of description logics. It is not expressed in XML but could easily be. It constrains the KRSS-compatible processors to accept the complete (very expressive) set of KRSS constructors and to raise a warning when some of them are not supported. It is thus not really modular in the sense that no specific sub-language is defined (a core language is defined in [9] but has no particular instanciation).

Conclusion

The DLML framework offers a general encoding of description logics in XML such that the modularity of description logics can be used in XML (for extending the language, building transformation stylesheets ...). Although it has been illustrated only by restricted examples, such an encoding has great potential for the interoperability of knowledge representation systems. In particular, it allows the implementation of the 'family of languages' approach to semantic interoperability which takes advantage of a group of comparable languages (here description logics) in order to select the best suited for a particular task. We are currently experimenting the proof-carrying transformation idea in this context.

Some of the work described here is accessible from the web site `http://co4.inrialpes.fr/xml/dlml/`.

References

[1] Sean Bechhofer, Ian Horrocks, Peter Patel-Schneider, and Sergio Tessaris. A proposal for a description logic interface. In *Proc. Description logics workshop, Linkoping (SE)*, number CEUR-WS-22, 1999. http://SunSITE.Informatik.RWTH-Aachen.DE/Publications/CEUR-WS/Vol-22/bechhofer.ps.

[2] Tim Berners-Lee. Semantic web roadmap, 1998. http://www.w3.org/DesignIssues/Semantic.html.

[3] Alexander Borgida. Extensible knowledge representation: the case of description reasoners. *Journal of artificial intelligence research*, 10:399–434, 1999.

[4] Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen (eds.). Extensible Markup Language (XML) 1.0. Recommendation, W3C, 1998. http://www.w3.org/TR/REC-XML/.

[5] Francesco Donini, Maurizio Lenzerini, Daniele Nardi, and Andrea Schaerf. Deduction in concept languages: from subsumption to instance checking. *Journal of logic and computation*, 4(4):423–452, 1994.

[6] James Clark (ed.). XSL transformations (XSLT) version 1.0. Recommendation, W3C, 1999. http://www.w3.org/TR/xslt.

[7] Dieter Fensel, Ian Horrocks, Frank Van Harmelen, Stefan Decker, Michael Erdmann, and Michael Klein. Oil in a nutshell. In *12th International Conference on Knowledge Engineering and Knowledge Management EKAW 2000*, Juan-les-Pins, France, 2000.

[8] George Necula and Peter Lee. Efficient representation and validation of proofs. In *Proceedings of the 13th LiCS, Indianapolis (IN US)*, pages 93–104, 1998.

[9] Peter Patel-Schneider and William Swartout (eds.). Description-logic knowledge representation system specification, 1993.

[10] Lynn Andrea Stein, Dan Connolly, and Deborah McGuinness (eds.). Daml-ont initial release, 2000.

[11] Henry Thompson, David Beech, Murray Maloney, and Noah Mendelsohn (eds.). XML Schema part 1: structures. Recommendation, W3C, 2001. http://www.w3.org/TR/xmlschema-1/.