

An infrastructure for formally ensuring interoperability in a heterogeneous semantic web

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

Abstract. Because different applications and different communities require different features, the semantic web might have to face the heterogeneity of the languages for expressing knowledge. Yet, it will be necessary for many applications to use knowledge coming from different sources. In such a context, ensuring the correct understanding of imported knowledge on a semantic ground is very important. We present here an infrastructure based on the notions of transformations from one language to another and of properties satisfied by transformations. We show, in the particular context of semantic properties and description logics markup language, how it is possible (1) to define properties of transformations, (2) to express, in a form easily processed by machine, the proof of a property and (3) to construct by composition a proof of properties satisfied by compound transformations. All these functions are based on extensions of current web standard languages.

1. Introduction

The idea of a “semantic web” [Berners-Lee 2001] supplies the (informal) web as we know it with annotations expressed in a machine-processable form and linked together. Taking advantage of this semantic web will require the manipulation of knowledge representation formalisms.

There are several reasons why the semantic web could suffer from diversity and heterogeneity. One main reason is that it depends on content providers and content providers have diverse goals and focal points that will not lead them to invest on the same area of the semantic web. Yet these areas of interest will overlap meaningfully and putting part of their content together will be required for taking advantage of them in unexpected applications [Wiederhold 1999]. Another reason arises from the observation that the web sites and web pages are more often generated on demand depending on (1) the device on which they will be displayed and (2) the preferences of the users. There is no reason why the semantic web resources would not require the same kind of operations. There are several other reasons for expecting heterogeneity including legacy knowledge bases and systems, learning curves...

Because we think that nothing better can happen to the semantic web than having well suited languages for each task while preserving interoperability, we aim at providing a path toward this goal. This paper is a short description of the technicalities involved in a solution to interoperability despite diversity.

Imagine a second-hand hardware provider company willing to build a semantic web support for its business involving repair and printers. Because the company core competence is neither technical support, nor printers, it will prefer to reuse knowledge models (or ontologies, which can be quickly described as conceptual schemes of knowledge bases) from authoritative sources. Additionally, the company has decided to use a particular representation and deduction formalism for processing knowledge (similar to the SHIQ language for which the FaCT reasoner can perform subsumption test). This company will find a technical support ontology written in DAML-ONT and a printer ontology written in OIL that fulfill its requirements.

The problem then consist of importing these two ontologies in the SHIQ language in a semantically valid way (i.e. preserving the consequences they entail). The solution will resort to transforming each ontology into a common format and transforming this format into a form compatible with SHIQ. This can be achieved by a homemade transformation or by assembling transformations available through the web (see Figure 1). Of course, the transformation system engineer will choose transformations that satisfy the desired properties (consequence preservation). To that extent, (s)he will refer to the properties advertised for each transformation. But (s)he has to make sure that the assertions are correct (they can be erroneous, or valid within a specific context...). There are two basic alternatives to this problem: trusting or checking.

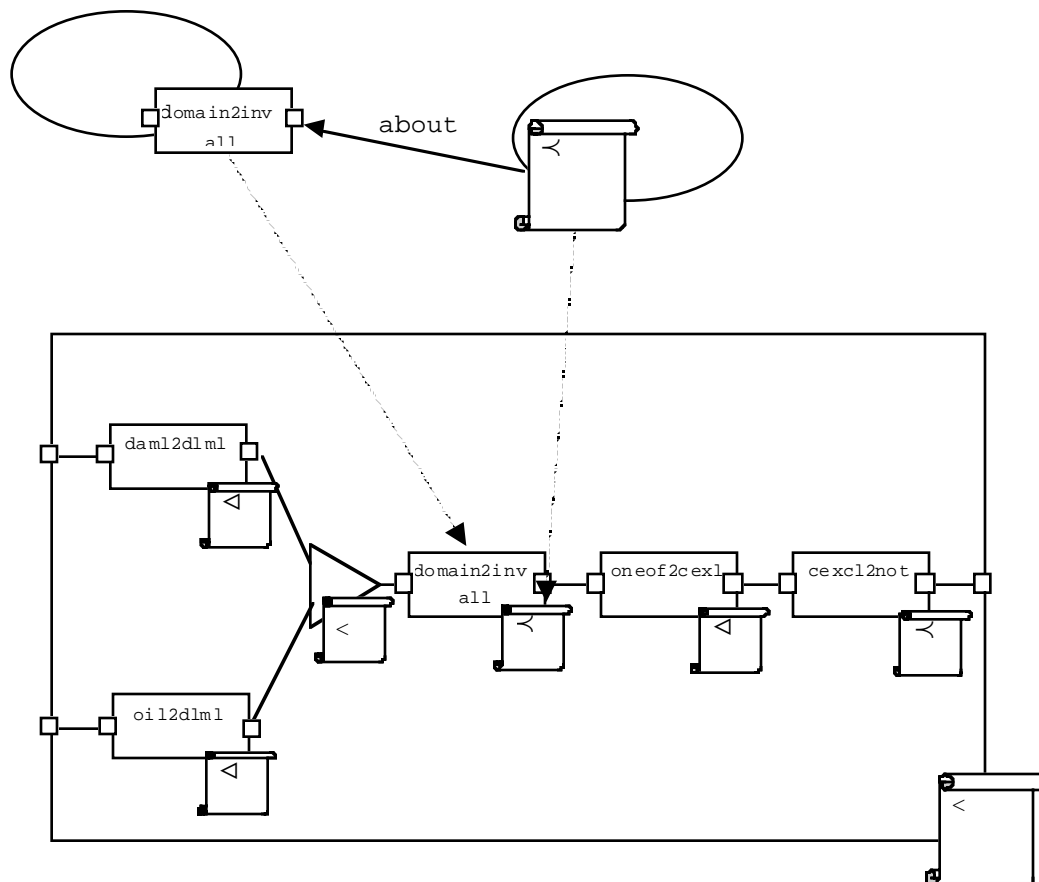


Figure 1 : The complete construction of a transformation, by composing more elementary transformations gathered from the web, and the proof of consequence preservation by composing lemmas.

Checking is possible if the proofs of the asserted properties are available somewhere. It will then be possible to check the properties satisfied by the transformations and to deduce those satisfied by the compound transformations. In order to contribute to the global web of transformations, the transformation system engineer will publish the new compound transformation and the proofs of its properties.

The framework presented here is distributed, modular, incremental (i.e. anyone can add a new transformation, a new assertion or a new proof at any time) and ensure a high level of safety. In these matters, it is fully adequate for the semantic web.

The remainder presents the building blocks of such an infrastructure. The presentation is based on the simple example above (the complete example has been implemented in DLML and XSLT). First, we describe DLML, an XML encoding of knowledge representation language and the kind of transformations that can be performed on these languages (§2). Then several consequence-preserving properties are introduced (§3). The proofs of such properties are expressed in such a way that machines can manipulate them (§4). Last, we

introduce an environment for building, checking, proving and publishing transformation and proofs by composition (§5).

2. A family of representation languages : DLML

In order to simplify the presentation and to facilitate the transformations, we will restrict ourselves to a set of languages that act as pivot languages between the actual representation languages used in the semantic web.

In this presentation, a language L will be a set of expressions. A representation (r) is a set of expressions in L . In this framework, a model of a set of assertions $r \subseteq L$, is an interpretation I satisfying all the assertions in r . An expression δ is said to be a consequence of a set of expression r if it is satisfied by all models of r (this is noted $r \models_L \delta$). A family of languages is a set L of languages that share constructors having the same interpretation in all the languages. A family can be structured such that a language $L \vee L'$, such that any formula of L or L' is a formula of $L \vee L'$, always exists. The “family of languages” approach [Euzenat 2001c] is an interesting case, because it enables a fast implementation of meaning-preserving transformations. Using a family of languages makes the representations easier to understand because the elements have the same meaning across languages. It will enable the fragmentation of these transformations into unit transformations and the precise characterization of the transformation properties.

A good example of a family of languages is the description logics for which an extensive language hierarchy has been defined [Donini 1994]. This presentation will focus on our “Description Logic Markup Language” (DLML) on which we have carried out experiments. DLML [Euzenat 2001d] is a modular system of document type descriptions (DTD) encoding the syntax of many description logics (§2.1). The actual system contains the description of more than 40 constructors and 25 logics. To DLML is associated a set of transformations (written in XSLT) enabling the conversion of a representation from one logic to another (§2.2).

Note that we do not put forth DLML as the standard language of the semantic web but rather as one of the many languages that can be used for transformation purposes. DLML is used here as a proof of concept. The general framework, however, will work with other languages.

2.1 Modular Encoding

Description logics allow the manipulation of two kinds of terms: concepts and roles. Below are one role description stating that the role `inktype` has for domain of application the `InkPrinter` concept and one concept term description stating that a `ColorInkPrinter` is an `InkPrinter` whose `inktype(s)` are all instances of the `ColorInkType` concept.

```
inktype ≤ (domain InkPrinter)
ColorInkPrinter ≤ (and InkPrinter (all inktype ColorInkType))
```

Term descriptions are built from sets of atomic concept (resp. role) names and term constructors. They are constrained by equations of the kind above where two terms are related by a formula constructor (here \leq). A terminology is a set of such equations.

Concept terms are interpreted as sets of individuals of the domain of interpretation and roles are sets of pairs of individuals. The interpretation I of the constructors above is :

$$\begin{aligned}
 I((\text{and } c_1, \dots, c_n)) &= I(c_1) \cap \dots \cap I(c_n) \\
 I((\text{all } r \ c)) &= \{ x \in D ; \forall y ; \langle x, y \rangle \in I(r) \Rightarrow y \in I(c) \} \\
 I((\text{inv } r)) &= \{ \langle x, y \rangle ; \langle y, x \rangle \in I(r) \} \\
 I((\text{domain } c)) &= \{ \langle x, y \rangle ; x \in I(c) \}
 \end{aligned}$$

As usual, a model of a terminology is an interpretation I which satisfies all the assertions of the terminology.

DLML takes advantage of the modular design of description logics by describing individual constructors separately. The modular encoding of the description logics is made of three kind of DTD: atoms (introducing the atomic terms), term constructors (e.g., `all`, `and`, `not`) and formula constructors (e.g. `=`, `≤`). An arbitrary number of these XML 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 dl:INV (%dl:RDESC;)>
```

We have also defined the notion of Document Semantic Description (DSD) which enables the description of the formal semantics of an XML language (just like the DTD or schemas express the syntax). To the DTD above is associated a DSD describing the semantics of the operator (i.e. $I((\text{inv } r)) = (I(r))^{-1}$):

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

In the experimental DSD language, the XML elements are identified by XPATH [Clark 1999b] expressions (`dl:INV` or `*[1]` standing for any term of constructor `INV` and any first argument of the term). The syntax is very similar to that of XSLT [Clark 1999a] (with `denotation`, `interpretation` and `apply-interpretation` corresponding to `template` and `apply-template`). The remaining expressions are mathematical symbols expressed in MathML [Carlisle 2001].

The DLML family of languages provides the DTD and DSD of all the covered operators and can build automatically those of a particular logic from its DLML description. The DLML logic descriptions are like the following:

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

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

  <dlml:cop name="anything"/>
  <dlml:cop name="nothing"/>
  <dlml:cop name="and"/>
  <dlml:cop name="or"/>
  <dlml:cop name="not"/>
  <dlml:cop name="all"/>
  <dlml:cop name="some"/>
  <dlml:cop name="csome"/>
  <dlml:cop name="catleast"/>
  <dlml:cop name="catmost"/>
  <dlml:rop name="inv"/>
  <dlml:rop name="trans"/>

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

From this description, two XSLT stylesheets can generate the DTD and DSD corresponding to the language. They can be used for expressing SHIQ terminologies in XML.

2.2 Transformations

What can such a DTD for description logics be good for? Once a language is encoded in XML, it is very easy to transform syntactically a representation into another one. A transformation is an algorithmic manner to generate one representation from another (not necessarily in the same language). A transformation $\tau:L\rightarrow L'$, from a representation r of L generates a representation $\tau(r)$ in L' .

More precisely, we take advantage of the XSLT transformation language (“XML Style Language Transformations” [Clark 1999a]) recommended by W3C, for which we put forward a compound transformation language (see §5.1).

The first application is the import and export of terminologies from a description logic. In our example, the representations in OIL and DAML-ONT are imported in DLML through transformations. Then, the result is exported to SHIQ (the FaCT system [Bechhofer1999a] has an XML entry point). These transformations are simple XSLT stylesheets.

More elaborate transformations can be developed. The imported representations are then merged and three successive steps (inspired from those of OIL [Horrocks 2000]) are applied to the result: the three steps concern the suppression of the `DOMAIN` constructor with the help of the `ALL` and `INV` constructors (`domain2allinv`), the suppression of the `ONE-OF` constructor with the help of new exclusive concepts (`oneof2orcexcl`) and the elimination of the exclusion introducers with the help of the `NOT` constructor (`cexcl2not`).

The piece of stylesheet presented below converts a terminology containing the `DOMAIN` restrictions on roles (attributes) in a terminology which replaces them by a `ALL` constraint on the inverse (`INV`) of the role applied on the whole universe (`ANYTHING`). For instance, it will convert:

```
inktype ≤ (domain InkPrinter)
```

into:

```
Anything ≤ (all (inv inktype) InkPrinter)
```

Both formulas equally say that only InkPrinters can have the inktype attribute.

```
<xsl:template match="dl:TERMINOLOGY">
  <dl:TERMINOLOGY>
    <xsl:comment>Introduction of the DOMAIN</xsl:comment>
    <dl:CPRIM>
      <dl:ANYTHING />
      <dl:AND>
        <xsl:apply-templates select="dl:RPRIM " mode="gatherdomain" />
      </dl:AND>
    </dl:CPRIM>
    <xsl:comment>The terminology</xsl:comment>
    <xsl:apply-templates />
  </dl:TERMINOLOGY>
</xsl:template>

<!-- gather domains in role introduction and add this for root -->

<xsl:template match="dl:RPRIM " mode="gatherdomain">
  <dl:ALL>
    <dl:INV>
      <dl:RATOM><xsl:value-of select="dl:RATOM[1]/text()" /></dl:RATOM>
    </dl:INV>
    <xsl:apply-templates select="dl:DOMAIN/*" />
  </dl:ALL>
</xsl:template>

<!-- usual processing -->
```

```

<xsl:template match="*|@*|text()">
  <xsl:copy><xsl:apply-templates select="*|@*|text()"/></xsl:copy>
</xsl:template>

<xsl:template match="dl:RPRIM">
  <dl:RPRIM>
    <dl:RATOM><xsl:value-of select="dl:RATOM[1]/text()"/></dl:RATOM>
    <xsl:choose>
      <xsl:when test="dl:DOMAIN">
        <dl:ANYRELATION/>
      </xsl:when>
      <xsl:otherwise><xsl:copy-of select="."/></xsl:otherwise>
    </xsl:choose>
  </dl:RPRIM>
</xsl:template>

```

This stylesheet gathers all the `DOMAIN` constraints of relations in a range (`ALL`) constraint of the inverse (`INV`) of the relation and applies it to `ANYTHING`. Then, it reproduces the whole terminology with domain constraints suppressed (i.e. replaced by `ANYRELATION`)¹.

Such transformations are assembled for transforming terminologies in one logic into another, equivalent, one. This is what is achieved in our example involving the three transformations.

3. Properties : consequence preservation

Operationally, the content of the previous section is sufficient for importing a representation from one language to another. However, it does not provide any idea of what properties are satisfied by each transformation step, nor by the transformation process as a whole. In order for the semantic web to be safely used by machines, it is necessary to define what properties have to be satisfied by the transformations. We focus here on the consequence preservation property (a semantic property) which is described in §3.1. In the context of families of languages we have described a set of more precisely characterized properties that entail consequence preservation. They are presented in the following subsections (§3.2-3.4).

3.1 Transformation properties

A property is a Boolean predicate about the transformation (e.g., “preserving information” is such a predicate — it is true or false of a transformation — and is satisfied if there exists an algorithmic way to recover r from $\tau(r)$). We consider more closely preservation properties which preserve (or counter-preserve) an order relation between the source representation (r) and the target representation ($\tau(r)$). There can be many such properties (content or structure preservation, traceability, and confidentiality...) affecting different aspects of the representation. They can be roughly classified as:

- Syntactic properties : like the completion ($\tau(r) \ll r$, in which \ll denotes structural subsumption between representations) ;
- Semantic properties : like consequence preservation ($\tau(r) \Rightarrow r$, i.e. equation 2 below) ;
- Semiotic properties : like interpretation preservation (let σ be the interpretation rules and \models_i be the interpretation of individual i , $\forall \delta \in L, \forall i, j, r, \sigma \models_i \delta \Rightarrow \tau(r), \tau(\sigma) \models_j \tau(\delta)$).

¹ This transformation is not sufficient to eliminate all occurrences of domain. For instance, (all (domain C) C') has to be transformed into (or (not C) (all anyrelation C')). But this is sufficient for our demonstration.

In the context of the communication of formal representations, we would like to warrant the preservation of the meaning of the representations. This can be defined by the two complementary equations:

$$\forall r \subseteq L, \forall \delta \in L, r \models_L \delta \Rightarrow \tau(r) \models_{L'} \tau(\delta) \quad (1)$$

$$\forall r \subseteq L, \forall \delta \in L, \tau(r) \models_{L'} \tau(\delta) \Rightarrow r \models_L \delta \quad (2)$$

Generalized interoperability is, of course, out of reach. Consequently we study restricted cases of these equations. In the context of the “family of languages” approach we identified several properties presented below which are more precise and entail equation (1).

3.2 Language inclusion

The simplest transformation is the transformation from one logic to a syntactically more expressive one (i.e. which adds new constructors). The transformation is then trivial, but yet useful, because the initial representation is valid in the new language; it is thus identity:

$$\forall \delta \in L, r \models_L \delta \Rightarrow r \models_{L'} \delta$$

This trivial interpretation of semantic interoperability is one strength of the “family of languages” approach because, in the present situation, nothing has to be done for gathering knowledge. For this case, one can define the relation between two languages L and L' as $L < L'$ which has to comply with $L \subseteq L'$. We can then define $L = L'$ as equivalent to $L < L'$ and $L' < L$. This defines the syntactic structure of L .

This simple property is satisfied by the merge operation that puts together the two representations issued from the DAML-ONT translation and the OIL translation.

3.3 Model preservation

If $L < L'$ does not hold, the transformation is more difficult. The initial representation r can be restricted to what is (syntactically) expressible in L' : $\tau_{\prec}(r)$. However, this operation (which is correct) is incomplete because it can happen that a consequence of a representation expressible in L' is not a consequence of the expression of that representation in L' :

$$\exists \delta \in L'; \tau_{\prec}(r) \not\models_{L'} \delta \text{ and } r \models_L \delta$$

To solve this problem, as stated in [Visser 2000a], it is necessary to deduce from r in L whatever is expressible in L' . Let $\tau_{\prec}(r) = \tau_{\prec}(\text{Cn}(r))$ be this expression. It is such that $\forall r \subseteq L, \forall \delta \in L \wedge L', r \models_L \delta \Rightarrow \tau_{\prec}(r) \models_{L'} \delta$.

The previous proposal is restricted in the sense that only expressions of the source language are allowed in the target language, though there exist equivalent non-syntactically comparable languages. This is the case of the description logic languages ALC and ALUE which are known to be equivalent while none has all the constructors of the other. For that purpose, one can define $L < L'$ if and only if the models are preserved, i.e.

$$\exists \tau_{\prec}; \forall r \subseteq L, \forall \langle I, D \rangle; \langle I, D \rangle \models_L r \Rightarrow \langle I, D \rangle \models_{L'} \tau_{\prec}(r)$$

This property is satisfied by the `domain2allinv` and `cexcl2not` transformations.

The τ_{\prec} transformation is not easy to produce (and can generally be computationally expensive) but we show, in §4.1, how this can be practically achieved.

3.4 Model isomorphism

Another possibility is to define \triangleleft as the existence of an isomorphism between the models of r and those of $\tau_{\triangleleft}(r)$:

$$\exists \tau_{\triangleleft}; \forall \langle I', D' \rangle, \exists \langle I, D \rangle; \forall r \subseteq L, \langle I', D' \rangle \models_{L'} \tau_{\triangleleft}(r) \Rightarrow \langle I, D \rangle \models_L r$$

This also ensures that $\forall r \subseteq L, \forall \delta \in L, r \models_L \delta \Rightarrow \tau_{\triangleleft}(r) \models_{L'} \tau_{\triangleleft}(\delta)$.

This property is satisfied by the `oneof2orcexcl` transformation. It can be used in order to use a prover built for a logic with a logic whose models are isomorphic to it.

This provides a structure based on semantics to the family of languages L . Summarizing, the syntactic and semantic structure of a language family provides different semantic properties characterizing transformations, all of them entailing consequence preservation.

We have considered only transformations that do preserve all information because languages have at least the same expressivity. It can happen that representations are imported to a language of lower expressivity. In such a case, consequence preservation cannot be ensured. Some information must be lost by the transformation. This can be subject to properties that characterize the kind of information that can be sacrificed.

4. Proofs, annotations and proof-checking

The approach to semantic interoperability defended here is based on transformations and their properties. Hence, in order to ensure formally the properties of transformations, one must exhibit a proof of the property. In fact, the proof and the transformation can be strongly tied together to the extent that they are built together (§4.1). In such a case, the publication of the proof is as important as the publication of the transformation (§4.2). The proof can be checked thus providing confidence with the corresponding transformation (§5.2).

4.1 From proofs to transformations

When providing transformations from one language to another, it is useful to prove the properties that are satisfied by the transformations (e.g. that the transformation terminates or that it preserves interpretations). For instance, the proof that the `domain2allinv` transformation preserves interpretations is as follows (inference rules are in brackets):

$$\begin{aligned}
& r \leq (\text{domain } C) && \text{[hypothesis]}(0) \\
\Rightarrow & I(r) \subseteq I((\text{domain } C)) && \text{[dsd/syn-to-sem]}(1) \\
\Rightarrow & I(r) \subseteq \{\langle x,y \rangle \in D^2; y \in I(C)\} && \text{[dsd/expand-interp]}(2) \\
\Rightarrow & \forall \langle x,y \rangle \in I(r), y \in I(C) && \text{[sets/incl-in]}(3) \\
\Rightarrow & \forall x \in D, \forall y, \langle x,y \rangle \in I(r) \Rightarrow y \in I(C) && \text{[pc/quant-intro]}(4) \\
\Rightarrow & \forall x \in D, \forall y \langle x,y \rangle \in \{\langle w,z \rangle; \langle z,w \rangle \in I(r)\} \Rightarrow y \in I(C) && \text{[set/in-incl]}(5) \\
\Rightarrow & D \subseteq \{x \in D, \forall y; \langle x,y \rangle \in \{\langle w,z \rangle; \langle z,w \rangle \in I(r)\} \Rightarrow y \in I(C)\} && \text{[dsd/retract-interp]}(6) \\
\Rightarrow & D \subseteq \{x \in D; \forall y; \langle x,y \rangle \in I((\text{inv } r)) \Rightarrow y \in I(C)\} && \text{[dsd/retract-interp]}(7) \\
\Rightarrow & I(\text{Anything}) \subseteq \{x \in D; \forall y; \langle x,y \rangle \in I((\text{inv } r)) \Rightarrow y \in I(C)\} && \text{[dsd/retract-interp]}(8) \\
\Rightarrow & I(\text{Anything}) \subseteq I(\text{all } (\text{inv } r) C) && \text{[dsd/retract-interp]}(9) \\
\Rightarrow & \text{Anything} \leq (\text{all } (\text{inv } r) C) && \text{[dsd/sem-to-syn]}(10)
\end{aligned}$$

This proof, like many language equivalence proofs in description logics, shows that whatever term built from some term constructor (here `DOMAIN`) is expressible with other term constructors (here `ALL`, `INV` and `ANYTHING`) though preserving the interpretation of the terms. One characteristic of such proofs in term-based languages is that they are constructive: they exhibit a transformation from one language to the other. They can thus be translated into a transformation (and this results in the XSLT example presented in §2.2).

Another example is the transformation from `ALUE` to `ALC`, which is based on the argument that any `NOT` constructor can be pushed down the term structure:

$$\begin{aligned}
(\text{not } c) & \Leftrightarrow (\text{anot } c) && \text{for } c \text{ atomic} \\
(\text{not } (\text{anot } c)) & \Leftrightarrow c \\
(\text{not } (\text{not } c)) & \Leftrightarrow c \\
(\text{not } (\text{all } r c)) & \Leftrightarrow (\text{c some } r (\text{not } c)) \\
(\text{not } (\text{and } c_1, \dots, c_n)) & \Leftrightarrow (\text{or } (\text{not } c_1) \dots (\text{not } c_n))
\end{aligned}$$

$$(\text{not } (\text{some } r)) \Leftrightarrow (\text{all } r \text{ Nothing})$$

This proof can be turned into a transformation, which applies the rules (from left to right) recursively on the structure of the terms. In DLML, many of the transformations across languages have been designed together with their proofs. We did this for the above transformations. This principle, which is the instantiation of the Curry-Howard correspondence, can be applied to many transformations.

4.2 Proof annotations

If the designers build proofs of some properties, it is desirable, especially in a worldwide distributed environment, to publish these proofs. It is thus useful to be able to represent them. The representation of the proof itself can be provided in MathML [Carlisle 2001] and OMDoc [Kohlhase 2000] a language extending MathML towards the expression of mathematical macrostructures (e.g., theories, theorems, axioms, and proofs). In this formalism, the two first steps of the proof above would look like:

```
<omd:proof id='domain2allinvpr' for='domainelim' theory='dlml'>
  <omd:hypothesis id='domain2allinv_0' />
  <omd:derive id='domain2allinv_1'>
    <omd:FMP>
      <omd:assumption id='domain2allinv_0'>
        <OMOBJ>
          <dl:rprim>
            <dl:ratom>r</dl:ratom>
            <dl:domain>
              <dl:catom>C</dl:catom>
            </dl:domain>
          </dl:rprim>
        </OMOBJ>
      </omd:assumption>
      <omd:conclusion id='domain2allinv_1cl'>
        <OMOBJ>
          <mml:apply><mml:subset/>
            <dsd:apply-interpretation>
              <dl:ratom>r</dl:ratom>
            </dsd:apply-interpretation>
            <dsd:apply-interpretation>
              <dl:domain><dl:catom>C</dl:catom></dl:domain>
            </dsd:apply-interpretation>
          </mml:apply>
        </OMOBJ>
      </omd:conclusion>
    </omd:FMP>
    <omd:method><omd:ref theory='dsd' name='syn-to-sem' /></omd:method>
    <omd:premise xref='domain2allinv_0' />
  </omd:derive>
  <omd:derive id='domain2allinv_2'>
    <omd:FMP>
      <omd:assumption id='domain2allinv_1cl' />
      <omd:conclusion id='domain2allinv_2cl'>
        <OMOBJ>
          <mml:apply><mml:subset/>
            <dsd:apply-interpretation>
              <dl:ratom>r</dl:ratom>
            </dsd:apply-interpretation>
            <dsd:apply-interpretation>
              <dl:domain><dl:catom>C</dl:catom></dl:domain>
            </dsd:apply-interpretation>
          </mml:apply>
        </OMOBJ>
      </omd:conclusion>
    </omd:FMP>
  </omd:derive>
</omd:proof>
```

```

        </OMOBJ>
    </omd:conclusion>
</omd:FMP>
<omd:method><omd:ref theory='dsd' name='expand-interp'/></omd:method>
<omd:premise xref='domain2allinv_1cl'/>
<omd:conclusion>
</omd:derive>
...
<omd:derive id='domain2allinv_10'>
  <omd:FMP>
    <omd:assumption id='domain2allinv_9cl'/>
    <omd:conclusion id='domain2allinv_10cl'>
      <OMOBJ>
        <dl:cprim>
          <dl:anything/>
          <dl:all>
            <dl:inv><dl:ratom>r</dl:ratom></dl:inv>
            <dl:catom>C</dl:catom>
          </dl:all>
        </dl:cprim>
      </OMOBJ>
    </omd:conclusion>
  </omd:FMP>
  <!-- this is substitution of interpretation by its definition -->
  <omd:method><omd:ref theory='dlml' name='completeness'/></omd:method>
  <omd:premise xref='domain2allinv_9cl' >
</omd:derive>
<omd:conclude id='domain2allinv_10'>
  <omd:FMP>
    <omd:assumption id='domain2allinv_9cl'/>
    <omd:conclusion id='domain2allinv_10cl'>
      <OMOBJ>
        <dl:cprim>
          <dl:anything/>
          <dl:all>
            <dl:inv><dl:ratom>r</dl:ratom></dl:inv>
            <dl:catom>C</dl:catom>
          </dl:all>
        </dl:cprim>
      </OMOBJ>
    </omd:conclusion>
  </omd:FMP>
  <!-- this is substitution of interpretation by its definition -->
  <omd:method><omd:ref theory='dlml' name='completeness'/></omd:method>
</omd:conclude>
</omd:proof>

```

The namespace prefix are `omd` for OMDoc, `mm1` for MathML, `dsd` for DSD and `dl` for DLML. We took some liberty with OMDoc (e.g. instead of OpenMath objects — OMOBJ — we put MathML expressions, because DSD is based on MathML instead of OpenMath). However, this is just a matter of syntax: the relevant part is the ability of OMDoc for representing proofs.

It is also useful to attach the property and the proof to the transformations. One solution consists of adding it to the transformation structure. There are two problems with this solution: the XSLT language does not enable this, though Transmorpher does, and this would prevent people who are not owner of the transformation to claim properties and publish proofs. Hence the best solution seems to use RDF for annotating the transformations from the outside.

5. Composing transformations, composing proofs

In a family of languages, composing transformations can be a very convenient way to transform from one language to another. This is what has been proposed in the introductory example. Each elementary transformation can be used in various compound transformations. We have developed a system, Transmorpher, for dealing with such composition of more elementary transformations (called transformation flows, §5.1). Transmorpher is an environment for defining transformations and assembling them, on one hand, annotating them by properties they satisfy or those they must satisfy and proving the properties of compound transformations on the other hand. The proof of properties of components can be gathered from the web and checked (§5.2) and the proof of the compound transformation can be obtained by composing the properties of the components (§5.3). Once the proofs are produced, both the transformation and the proof can be exported to the web (§5.4).

5.1 *Transmorpher*

In order to prove or check the properties of transformations, it is necessary to have a representation of these transformations. The XSLT language enables the expression of a transformation in XML but is relatively difficult to analyze. In order to overcome that problem, we have designed and developed in collaboration with the FluxMedia company, the Transmorpher environment [Euzenat 2001b]. It is a layer on top of XSLT allowing the expression of complex transformation flows such as the one of Figure 2 (which is that of the example). A transformation flow is the composition of elementary transformation instances whose input/output are connected by channels. A transformation flow is itself a transformation.

One of the goals of Transmorpher is the encapsulation of XSLT, used for performing the transformations, such that transformations are easier to analyze through special purpose syntax and hierarchical decomposition. This should facilitate the description of proofs through “Lemmas” attached to component transformations.

Transmorpher enables the definition and processing of generic transformations of XML documents. It provides XSLT extensions for:

- Describing straightforwardly simple transformations (removing elements, replacing attribute names, merging documents...);
- Composing transformations by connecting their (multiple) input and output;
- Applying transformations until closure;
- Applying regular expression substitution;
- Calling external transformation engines (such as XSLT).

Transmorpher describes the transformation flows in XML. Input/output channels carry the information, mainly XML, from one transformation to another. Transformations can be other transformation flows or elementary transformations. Transmorpher provides a set of abstract elementary transformations (including their execution model) and one default instantiation. Among elementary transformations are external calls (e.g. XSLT), dispatchers, serializers, query engines, iterators, mergers, generators and rule sets. Figure 2 presents the representation of the above transformation flow in Transmorpher graphic format.

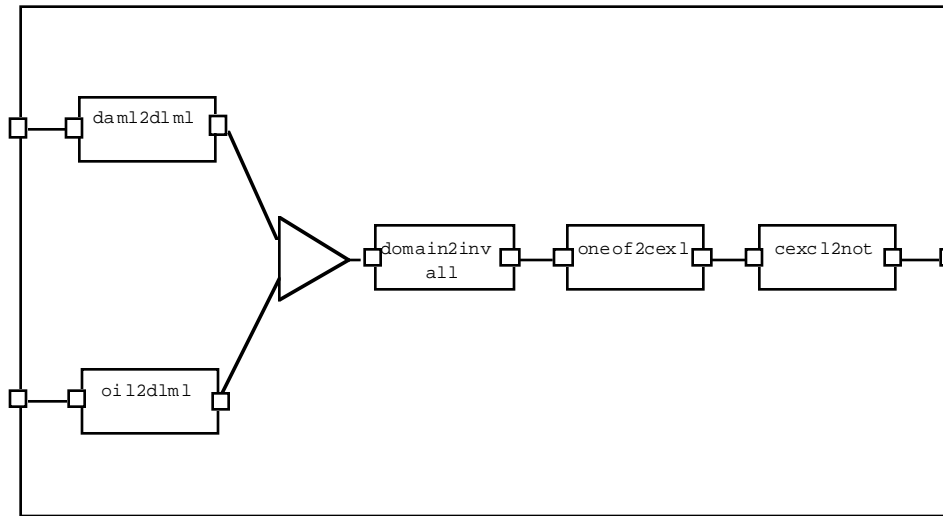


Figure 2 : Transmorpher description of the importation of DAML-ONT and OIL fragments into the DLML representation of SHIQ.

Transmorpher is mainly a set of documented Java classes (which can be refined or integrated into other software) and a transformation flow processing engine. A transformation flow can be expressed by programming in Java or providing an XML description. Figure 2 is the description of the following transformation flow:

```

<process name="assemble-onto" in="i1 i2" out="o">
  <apply-external type="xslt" name="daml2dlml" file="daml2dlml.xsl"
    in="i1" out="o1"/>
  <apply-external type="xslt" name="oil2dlml" file="oil2oil.xsl"
    in="i2" out="o2"/>
  <merge type="concat" name="ldaml+ldaml" in="o1 o2" out="o3"/>
  <apply-external type="xslt" name="domain2allinv"
    file="domain2allinv.xsl" in="o3" out="o4"/>
  <apply-external type="xslt" name="oneof2cexclor" file="oneof2or.xsl"
    in="o4" out="o5"/>
  <apply-external type="xslt" name="cexcl2not" file="cexcl2not.xsl"
    in="o5" out="o"/>
</process>

```

An extension of Transmorpher consists of attaching assertions to the transformations in a transformation flow in order to tell if a property is assumed, proved or to be checked. This will allow real experimentation of proving properties of compound transformations.

5.2 Towards proof checking

Proof-carrying code [Necula 1998] is an infrastructure in which a program is provided with the proof of the properties that it satisfies. A client system that wants to run the former program will check the proof against this program in order to ensure that it can do it safely. These principles can be applied to the verification of the transformations and their properties as soon as a representation of the proof is available.

In order to be able to check proofs of semantic properties such as (1) or (2), it is necessary to have (a) the representation of the transformation which is provided by XSLT or by Transmorpher, (b) the semantics of the transformation language, (c) the representation of the semantics of the logics provided by their DSD and (d) the representation of the proof like the one described above. Of these elements, the only missing one is the representation of the semantics of XSLT. There are several attempts, however, to provide a semantics for XSLT fragments that can be used [Wadler 2000, Bex 2000]. Another path consists of defining a

transformation language simpler than XSLT but with a clean semantics. This is partly the case of Transmorpher.

Checking is the opposite of trusting. Both approaches have different advantages: trusting does not require to spend time checking the arguments while checking does not require to maintain a heavy model of trust and is independent of who provides the arguments. Proof-carrying code can be applied to untrusted items. So if someone needs particular transformations satisfying particular properties, she can try to find such transformations and proof of properties on the web and check them.

Unlike watermarking, proof-carrying code does not require any encoding of the transformation because it checks the proof against the program. The program can have been modified, if the checker finds that the proof is still valid, then this is all that is required. It is not even required that the proofs are provided with the program. In fact, someone can publish an automatic proof of the termination of the above transformation web site not connected to the DLML one and the proof-checker must be able to decide if the proof is valid or not.

5.3 Proof by composition

Once the properties of elementary transformations are available, either by checking, trusting or proving, an interesting point is the elaboration of the proof of properties for transformation flows.

If each of these more elementary transformations is annotated by the assertion of the properties it satisfies, the property concerning the compound transformation remains to be computed. A very simple example is the termination property on finite input that is preserved through composition, but not by iteration until saturation. Model preservation for its part is preserved through both composition and iteration.

This can be exemplified with the properties that have been considered in §3. It is possible to establish the properties of the composition of two transformations given their own properties. This yields (the very simple) Table 1.

	<	<	<	≤	∅
<	<	<	<	≤	∅
<	<	<	<	≤	∅
<	<	<	<	≤	∅
≤	≤	≤	≤	≤	∅
∅	∅	∅	∅	∅	∅

Table 1 : Composition table for the semantic relations on transformations (≤ is consequence preservation).

Table 1 shows that the transformation flow above, that assembles all transformations of the example, is indeed consequence preserving.

5.4 Safe transformation development cycle

The techniques presented here provide a framework in which transformations from one representation language to another are available from the network and proofs of various properties of these transformations are attached to them. It is noteworthy that transformations and proofs do not have to come from the same origin. They can even be produced by the application.

The transformation system engineer can gather these transformations and their proofs, check the proofs before importing them in the transformation development environment. She will then be able to create a new transformation flow and generate the proofs of the required

properties. Finally, she will be able to publish on the network the transformation and its proof.

Given two languages with their semantics, in order to transform representations in one language into representations in the other that satisfy some properties, the following transformation edition process (see Figure 1) can be attempted:

1. Fetching transformations that can help performing part of the task ;
2. Fetching assertions and proofs about these transformations ;
3. Checking the proof or trusting the assertions of properties about the transformations ;
4. Composing transformations into a global transformation that is supposed to do the transformation;
5. Proving that this composition preserves the properties that are required by the global transformation ;
6. Publishing transformation, assertions and proofs for others to use it.

Then, the problem proposed in introduction will be reduced to: gather available ontologies, create a safe transformation flow for importing them in the current knowledge processing environment and apply the transformation flow. The transformation flow can be applied at any time for updating the compound ontology and its properties will remain valid as long as the languages remain the same.

6. Conclusion

We have presented a framework for formally ensuring semantic interoperability in the semantic web. Interoperability is assured by transformations that have to satisfy some client-defined properties. The proof of properties are encoded in a machine-readable way so that the client can check them. Transmorpher enables the composition of these transformations into a more elaborate one whose proof of properties can be facilitated by simple composition of the properties of its components (either proof-checked or trusted).

If enough actors are interested in sharing transformations safely instead of developing again and again the same transformation, here is an architecture enabling its formal and modular realization. We strongly believe that there will be a strong interest in such a framework in the context of the growing use of XML and XML transformations inside and across companies. In fact, if semantic properties are more related to the semantic web, many other properties of general interest can be taken into account by this framework.

The main strength of the framework is not its sophistication, but rather its relative simplicity. Its distributed, modular and incremental characteristics make it adapted to the web. No doubt that it will not be practical in all cases, but it works for cases like the one presented.

This framework is very close to that of proof-carrying code [Necula 1998] of which it is an instantiation on particular programs and properties. Moreover it is fully based on widely available XML technologies (XML, XPATH, XSLT, MATHML, OMDoc, RDF) or local extensions (DLML, DSD, Transmorpher). For a description of complementary work on the topic of semantic interoperability (e.g. [Masolo 1999, Chalupsky 2000, Ciocoiu 2000]), see [Euzenat 2001a].

This infrastructure is a prospective framework for which many pieces are already available and several of them linked together. The main part of it, with the notable exception of proof-checking, has already been implemented as a proof of concept. The DLML framework is operational and several experiments have been made with XSLT transformations. Transmorpher is an ongoing work whose basic functions are operational. The OMDoc and DSD languages are available.

We have some examples of proof (mainly of model preservation or model isomorphism) in description logics that should be a very good first testbed for the application of these concepts. We also have examples of transformations between heterogeneous representations (e.g. description logics and syllogistic).

The proof-checker is the difficult point because we will need one that can interface easily with the kind of proofs required by the framework. There are two issues to be solved next : generalization and scalability.

Generalization requires a lot of fundamental work about topics such as generalizing from DLML to other representation languages (we have superficially investigated syllogisms and considered DAML-ONT as a description logic language), generalizing semantics properties, generalizing to other (e.g. structural, semiotic) properties, generalizing the kind of proofs required. We are currently committed to investigate the semantic properties more thoroughly.

Robustification and scalability will be required in order to consider the workability of the whole system. Positive elements are the intrinsic distribution of our framework and the fact that any element can be replaced by another with similar interface.

Acknowledgements

The author is indebted to Heiner Stuckenschmidt who proposed the introductory example for presentation of the “family of languages” approach. Anonymous reviewers are thanked for suggesting several improvements of the paper.

References

- [Berners-Lee 2001] Tim Berners-Lee, James Hendler, Ora Lassila, The semantic web, *Scientific american* 279(5):35-43, 2001, <http://www.scientificamerican.com/2001/0501issue/0501berners-lee.html>
- [Bechhofer 1999] Sean Bechhofer, Ian Horrocks, Peter Patel-Schneider, Sergio Tessaris, A proposal for a description logic interface, Proc. int. workshop on description logics, Linköping (SE), CEUR-WS-22, 1999 <http://SunSITE.Informatik.RWTH-Aachen.DE/Publications/CEUR-WS/Vol-22/bechhofer.ps>
- [Bex 2000] Geert Jan Bex, Sebastian Maneth, Frank Neveu, A formal model for an expressive fragment of XSLT, *Lecture notes in computer science* 1861:1137-1151, 2000
- [Carlisle 2001] David Carlisle, Patrick Ion, Robert Miner, Nico Poppelier (eds.), Mathematical markup language (MATHML) version 2.0, Recommendation, W3C, 2001 <http://www.w3.org/TR/MathML2>
- [Chalupsky 2000] Hans Chalupsky, OntoMorph: a translation system for symbolic knowledge, Proceedings of 7th international conference on knowledge representation and reasoning (KR), Breckenridge, (CO US), pp471-482, 2000
- [Clark 1999a] James Clark (ed.), XSL transformations (XSLT) version 1.0, Recommendation, W3C, 1999 <http://www.w3.org/TR/xslt>
- [Clark 1999b] James Clark, Stephen DeRose (eds.), XML path language (XPath) version 1.0, Recommendation, W3C, 1999. <http://www.w3.org/TR/xpath>
- [Ciocoiu 2000] Mihai Ciocoiu and Dana Nau, Ontology-based semantics, Proceedings of 7th international conference on knowledge representation and reasoning (KR), Breckenridge, (CO US), pp539-546, 2000 <http://www.cs.umd.edu/~nau/papers/KR-2000.pdf>
- [Donini 1994] Francesco Donini, Maurizio Lenzerini, Daniele Nardi, Andrea Schaerf, Deduction in concept languages: from subsumption to instance checking, *Journal of logic and computation* 4(4):423-452, 1994
- [Euzenat 2001a] Jérôme Euzenat, Towards a principled approach to semantic interoperability, Proc. IJCAI workshop on Ontologies and information sharing, Seattle (WA US), 2001 to appear
- [Euzenat 2001b] Jérôme Euzenat, Laurent Tardif, XML transformation flow processing, Proc. 2nd Extreme markup languages, Montréal (CA), 2001, to appear, <http://transmorpher.inrialpes.fr/paper>
- [Euzenat 2001c] Jérôme Euzenat, Heiner Stuckenschmidt, The 'family of languages' approach to semantic interoperability, submitted, 2001
- [Euzenat 2001d] Jérôme Euzenat, Preserving modularity in XML encoding of description logics, Proc. 13th description logic workshop, Stanford (CA US), 2001 to appear
- [Horrocks 2000] Ian Horrocks, A denotational semantics for Standard OIL and Instance OIL, 2000, <http://www.ontoknowledge.org/oil/download/semantics.pdf>

- [Kohlhase 2000] Michael Kohlhase, OMDoc : an open markup format for mathematical documents, SEKI report SR-00-02, Universität des Saarlandes, Saarebrücken (DE), 2000
<http://www.mathweb.org/src/mathweb/omdoc/doc/omdoc/omdoc.ps>
- [Masolo 2000] Claudio Masolo, Criteri di confronto e costruzione di teorie assiomatiche per la rappresentazione della conoscenza: ontologie dello spazio e del tempo, Tesi di dottorato, Università di Padova, Padova (IT), 2000
- [Necula 1998] George Necula, Compiling with proofs, PhD thesis, Carnegie Mellon university, Pittsburgh (PA US), 1998
- [Visser 2000a] Ubbo Visser, Heiner Stuckenschmidt, G. Schuster, Thomas Vögele, Ontologies for Geographic Information Processing, Computers in Geosciences, 2001, to appear <http://www.tzi.de/buster/papers/Ontologies.pdf>
- [Wadler 2000] Philip Wadler, A formal semantics of patterns in XSLT, *Markup technologies*, 1999 <http://www.cs.bell-labs.com/who/wadler/papers/xsl-semantics/xsl-semantics.pdf>
- [Wiederhold 1999] Gio Wiederhold, Jan Janninck, Composing diverse ontologies, Proc. 8th IFIP working group on databases working conference on database semantics, Rotorua (NZ), 1999 <http://www-db.stanford.edu/SKC/publications/ifip99.html>