Implementing the COntent INterchange (COIN) Approach Through Use of Semantic Web Tools

Mihai Lupu
Stuart Madnick

Working Paper CISL# 2008-09

March 2008

Composite Information Systems Laboratory (CISL)
Sloan School of Management, Room E53-320
Massachusetts Institute of Technology
Cambridge, MA 02142

Implementing the COntent INterchange (COIN) Approach Through Use of Semantic Web Tools

Mihai Lupu[1] and Stuart Madnick[2]

[1] Singapore-MIT Alliance, National University of Singapore
mihailup@comp.nus.edu.sg
[2] Sloan School of Management, Massachusetts Institute of Technology
smadnick@mit.edu

Abstract. The COntext INterchange (COIN) strategy is an approach to solving the problem of interoperability of semantically heterogeneous data sources through context mediation. The existing implementation of COIN uses its own notation and syntax for representing ontologies. More recently, the OWL Web Ontology Language is becoming established as the W3C recommended ontology language. A bridge is needed between these two areas and an explanation on how each of the two approaches can learn from each other. We propose the use of the COIN strategy to solve context disparity and ontology interoperability problems in the emerging Semantic Web both at the ontology level and at the data level. In this work we showcase how the problems that arise from context-dependant representation of facts can be mitigated by Semantic Web techniques, as tools of the conceptual framework developed over 15 years of COIN research.

1 Introduction

Making computers understand humans is, generously put, a hard task. One of the main reasons for which this is such a hard task is because even humans cannot understand each other all the time. Even if we all spoke the same language, there still exist plenty of opportunities for misunderstanding. An excellent example is that of measure units. Again, we don't even have to go across different names to find differences: in the US, a *gallon* (the so-called Winchester gallon) is approximately 3785 ml while in the UK, the "same" *gallon* is 4546 ml, almost 1 liter more. So when we find a piece of information in a database on cars, for instance, and we learn that a particular model has a fuel tank capacity of 15 gallons, how much gas can we actually fit inside, and, consequently, for how long can we drive without stopping at a gas station?

The answer to the previous problem comes easy if we know where we got the data from: if the information was from the US, we know we can fit inside 56.78 liters of gas, while if it comes from the UK, it is 68.19 - a difference of about 11 liters, with which a car might go for another 100 miles (or 161 km if the driver is not American or British).

Many more such examples exist (see [NAS] for a particularly costly one) and the reason for which they persist is mainly because it is hard to change the schema of relational databases that do not include the units of their measurements simply because when they were designed, they were designed for a single context, where everybody would know what the units are. With globalization, off-shoring, out-sourcing and all the other traits of the modern economical environment, those assumptions become an obstacle to conducting efficient business processes.

Even in the context of a purely-semantic web application, such as the Potluck [pot] project developed at the Computer Science and Artificial Intelligence Laboratory at MIT (CSAIL), contextual information is not explicitly approached. The user is allowed to mash-up together information from different sites, but it is not taken into account the fact that those different data sources may have different assumptions about an entire array of concepts. This paper shows how the COIN strategy can be implemented in this new environment and how it can contribute to it.

1.1 Semantic Web $\rightleftarrows$ COntext INterchange

Our current work acknowledges the successes that the Semantic Web community has achieved, particularly in the standardization of expressive new languages, and builds on top of that, providing methods to address the problem of *context mediation* or *context interchange*. The two areas, Semantic Web research and Context Mediation research, are complementary to each other. Each one provides the means for, and, at the same time, enhances the other. In particular, context mediation research helps resolve semantic heterogeneity in OWL/RDF/XML data, while semantic web research provides the standards for ontology representation and reasoning. Figure 1 depicts this mutually beneficial environment.

1.2 Approach overview

Semantic web tools rely heavily on mathematical logic to perform inferences. The result of this, in combination with our desire to maintain a 100% pure logic approach, is that *facts* cannot be deleted or modified. For instance, even if we define a relation *hasName* to be of functional type (i.e. have a unique object for each subject), it is still legal to have two entries with different objects, such as `(location1,hasName,'London')` and `(location1,hasName, 'Londres')`, the conclusion of which will be that the names "London" and "Londres" denote the same location. This has both advantages and disadvantages which we will not discuss here, but refer the reader to a wealth of literature on mathematical logic. Instead, we focus on how we model context in this framework.

Continuing the automobile-related example from the previous section, let us imagine a scenario where we are a British individual looking to purchase a car, and one of our main concerns is the environment, so we want a car that has a low gas consumption. Of course, we would prefer a more sporty car, if possible.

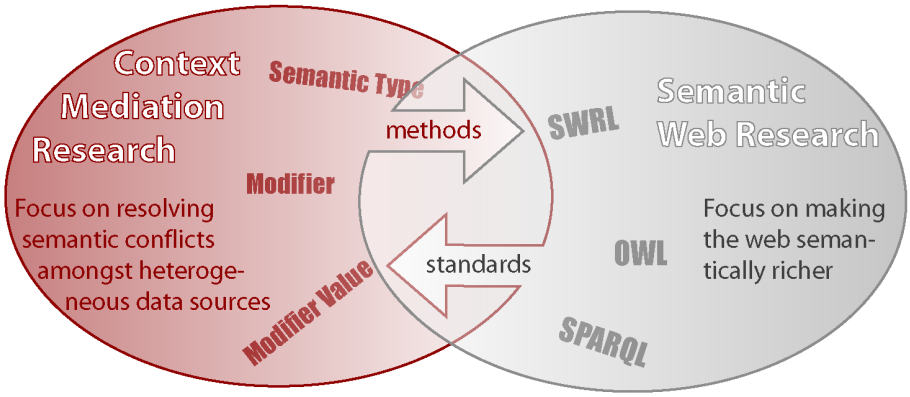


Fig. 1. Interaction between Semantic Web research and Context Mediation Research

We use the wealth of information on the Internet to do this, either via a mash-up tool like Potluck [pot], or just by browsing different car manufacturers' websites.

Imagine we look at a Ford Focus on `www.ford.com`, a mid-size car, and see that it does 24 mpg (miles per gallon). Since we are also interested in sports cars, we look at the Lotus, on `www.grouplotus.com`, and see that an introductory model does 25 mpg. We might then be convinced to spend the extra money to save the environment while enjoying the thrills of a true sports car. Unfortunately for us, it's not quite like that. We have the same misunderstanding that we mentioned before: the gallon. While the Focus considers an American gallon, the Lotus uses the British gallon. So if we transform the 24mpg of the Focus into British gallons, we get 28.8mpg - tempering our enthusiasm for the sports car.

This example is anecdotal but at the same time characteristic of the problems that occur when bringing together, mechanically, information residing in different databases.

We had previously analyzed a "Weather" example, where temperature units were converted automatically between Celsius and Fahrenheit [LM07]. Here, we extend that example and add a twist to it: now, the units are no longer different in their notation: *mpg* simply means different thing if we are in the US or in the UK.

Listing 1.1 shows an existing piece of information regarding the gas consumption of a car, while Listing 1.2 shows how we might represent different values of the same mileage using prefixes of relations.

Listing 1.1. Existing data regarding the gas consumption of a car

```
1 <?xml version="1.0"?>
2 <rdf:RDF
3   <US:mileage rdf:ID="mileage1">
```

```
    <US:hasValue rdf:datatype="[...]#float">
    24</US:hasValue>
  </US:mileage>
  <US:Automobile rdf:ID="Focus">
    <US:hasName rdf:datatype="[...]#string">
      Ford Focus</US:hasName>
    <US:hasMileage rdf:resource="#mileage1"/>
  </US:Automobile>
</rdf:RDF>
```

Listing 1.2. A possible solution to representing context, by adding an additional *hasValue* relation to the `Mileage` object. It is simple and intuitive of the fact that we are dealing indeed with the same mileage.

```
<?xml version="1.0"?>
<rdf:RDF
  <US:mileage rdf:ID="mileage1">
    <US:hasValue rdf:datatype="[...]#float">
    24</US:hasValue>
    <UK:hasValue rdf:datatype=‘‘[...]#float’’>
    28.8</UK:hasValue>
  </US:mileage>
  <US:Automobile rdf:ID="Focus">
    <US:hasName rdf:datatype="[...]#string">
      Ford Focus</US:hasName>
    <US:hasMileage rdf:resource="#mileage1"/>
  </US:Automobile>
</rdf:RDF>
```

The way to add this new relation is by defining a SWRL [HPSB+04] rule such as the one in Listing 1.3 and then querying the results with a query in SPARQL [PS07] as in Listing 1.4.

Listing 1.3. SWRL rule that generates a value in the UK context

```
US:hasValue(?mileage, ?mileageValue) ∧
US:hasMileage(?car, ?mileage) ∧
swrlb:multiply(?product,?mileageValue,12) ∧
swrlb:divide(?newValue,?mileageValue,10)
→  UK:hasValue(?mileage, ?newValue)
```

Listing 1.4. SPARQL query that retrieves the value in the UK context

```
SELECT ?mileageValue
WHERE { ?l US:hasMileage ?mileage .
              ?l US:hasName "Ford Focus" .
              ?mileage UK:hasValue ?mileageValue
            }
```

The SWRL rule in Listing 1.3 simply states that in order to get the UK mileage from the US mileage we have to multiply the original value by 1.2. Since SWRL does not handle floating point values, we do this my a multiplication in line 3 and a division in line 4.

The query in Listing 1.4 first identifies an entity ℓ (names starting with ? represent variables) which has a particular name ("Ford Focus") and a mileage. It returns the UK value of the mileage of the object ℓ.

This all seems very intuitive. As always, the problems lie in the details: How do we determine a conflict of contexts? How do we identify the correct rule to be applied? How should the data and rules be organized into files? Do we apply the rule to the entire dataset thus generating massive amounts of new data, or should we just apply it to the subset being queried?

After providing some background in Sections 2 and 3, we introduce the basic representation of context using the Ontology Web Language (OWL [MvH04]) in Section 4 and present our conflict identification and resolution method in Section 5.

2 Background and Related Work

2.1 COntext INterchange

The idea of the COntext INterchange [GBMS99] system is to re-use massive amounts of data that already exist but that are incomplete due to design assumptions that omitted constants from the dataset. When two or more such datasets are put together, or queried together, what were implied constants in each of them become variables in the aggregated dataset and consequently needs to be added back in the data. This is in most cases unfeasible due to the rigidity of the data structures or simply due to the fact that the end user has no control over the repository where the data exists.

The core of the COntext INterchange approach is a context mediator that rewrites queries coming from a user context into a context-sensitive mediated query that addresses the differences in meaning between the receiver and the sources. Conceptually, the context mediator is structured around a *domain model* that consists of *semantic types*, *attributes* and *modifiers*.

A semantic type is, as the name indicates, a conceptual entity. For instance, in the *Automobile* example of Section 3, the column `mileage` has no meaning by itself until it is associated with a semantic type *Mileage*. The coincidence in names is just because humans created both entities, but it should be clear to the reader that the column could very well have been named `milespergallon` and the semantic type *ST2842*. The important difference between semantic types and the columns with which they are associated is that the semantic types come enriched with semantics and attributes. In this simple example, *Mileage* has only one attribute, *value*.

In turns, an attribute may come endowed with a modifier. Again, using the *Automobile* example, we can imagine that the *value* attribute of the *Mileage*

semantic type has a modifier *unit*. This is called a *modifier* because it changes the meaning of the attribute to which it refers - it modifies it. The *modifier value* could be *Kilometers per liter* (*kml*) or *Miles per gallon* (*mpg*). In our example, we will only consider *mpg*, but keep in mind that it will mean different things in different contexts - something that we will need take into consideration and to model.

COIN uses this architecture to automatically determine differences in contexts and resolve queries in a way that is easy to interpret correctly by the user, even if the data is expressed in a different context. Existing application include financial reporting and analysis, airfare and car-rental aggregators, etc. [Fir03].

2.2 A glance at the Semantic Web

Though the COIN methodology precedes the Semantic Web, and despite the many similarities in objectives and motivation, the two have developed mostly independently. The wide spectrum of tools that have been proposed by different research groups to achieve the targets of the original paper by Berners-Lee et al. [BLHL01] make a quick but complete summary virtually impossible. In this section we just look at the few tools that we identify to be "best", both in terms of the appropriateness for our own purposes, and also in terms of their acceptance and popularity within the Semantic Web community.

Clearly, one of the pillars of current Semantic Web research is the Web Ontology Language (OWL) [MvH04]. To query the data stored in OWL format, one could map it back to a relational database and query it with SQL or use a "native" query language such as SPARQL [PS07]. For most purposes, SPARQL can be translated back to SQL, but the advantage of it lies in being able to query directly the RDF graph that underlies any ontology. It has the status of *Working Draft* of the W3C since October 2006. The necessity of defining a new query language for tuples, such as SPARQL may be questionable at first glance, since SQL is also working with tuples, though represented in a different way, and XQuery, also developed within the W3C, addresses the problem of querying XML, of which OWL is but a flavor. In [Mel06], the author argues that though it is true that most data could be represented conceptually in RDF and expressed in either relational databases or basic XML and thus queried by either SQL or XQuery, SPARQL provides a much easier way of querying the RDF graph, making the entire development process, including debugging, more fluent.

After representation and query languages, the Semantic Web framework requires a *rule language* to make inferences on the existing data, thus enabling the creation of the smart agents described in the original Berners-Lee paper. Though SWRL [HPSB$^+$04] has gained most attention in the past few years, the language has not yet been standardized by the W3C and many different implementations exist, that rarely support the full specification, mainly because in that case the reasoning becomes undecidable. One of the most popular implementation is SWRLTab [swr] - an extension to the Protégé framework [pro], that uses mainly the Jess [jes] inference engine (though it could use other en-

gines too). Other implementations are: R2ML [r2m], Bossam [bos], Hoolet [hoo], Pellet [pel], KAON2 [kao] and RacerPro [rac].

3 From tables to information

The first step towards making the data understandable by different agents[3] performing their activities in different contexts is to understand the fact that we are only dealing with representations of concepts and facts. As we exemplified before, 15 = 56.78 = 12.49 if one is gallons (US), one is liters and one is gallons (UK). Consequently, it makes more sense to have an abstract concept representing this volume and attach to it the knowledge that it may be expressed in different ways.

A tempting way of moving information out of the restrictive relational database is to encode it using XML. Using a naïve method implemented in most database systems, this would result in, literally, a data dump. For instance, a simple table containing cars and mileage values (Table 1) can be expressed as in Listing 1.5.

Using XML does not solve our problem. As discussed in [Mad01], XML is not a silver bullet - it is just another way to express the data. It only provides a more flexible way, allowing us to add more meaning to it. A simple "data dump" from the relational database is not enough for two reasons: First, as we see in Listing 1.5, the file mixes together the structure of the data with the data itself. Conceptually, these are different and should be represented as such. Second, the data itself is stored as if to preserve the physical appearance of the table (i.e. a sequence of rows, each with a few columns) rather than to preserve its underlying meaning. It is thus clear that a different approach is needed.

Table 1. Sample relational table

Automobile	**Mileage**
Ford Focus	24
Lotus Elise S	25

Listing 1.5. XML representation of relational database

```
1 <?xml version="1.0"?>
2 <mysqldump xmlns:xsi="[...]XMLSchema-instance">
3 <database name="test">
4  <table_structure name="US cars">
5   <field Field="automobile" Type="varchar(20)" />
6   <field Field="mileage" Type="float(11)"  />
7  </table_structure>
```

[3] we prefer the term 'agents' to show that they can be either human end-users or other computer systems

```
  <table_data name="cars">
   <row>
    <field name="automobile">Ford Focus</field>
    <field name="mileage">24</field>
   </row>
[...]
  </table_data>
 </database>
</mysqldump>
```

There exist attempts to extract ontological information from relational tables [Ast04,LM04]. What we want here is not nearly as ambitious as in these works. For our purpose, we don't necessarily need to infer a full scale ontology from the data, but simply to express things that are the same as being the same and things that are different as being different. It sounds simple for a human being, but computers have serious difficulties in performing even this simple task.

The first thing we want to do is separate the structure of the representation from the data itself. Listing 1.6 shows how we can define an ontological structure to organize the data in the table. We use the term "ontological" simply because we use the ontology specification language, OWL, but one should not imagine a complex theory behind it: in this listing we simply state that we deal with two concepts (*Automobile* and *Mileage*) who are connected by a relationship *hasMileage*. The difference between this approach and the simple XML dump is that here *automobile* and *mileage* are regarded as concepts, rather than fields in a table. It is a subtle, but essential difference. Here, a particular instance of the *Automobile* class has a name, but is separate from its name. This distinction will allow us later to specify that *Ford Focus* is the same car as *Focus* and that 28.8 is the same mileage as 24 (one using British gallons and one using American gallons). This way, in the data file shown in Listing 1.7 we can define the abstract mileage *mileage1* and give it a value and then define the abstract car *Focus* and give it a name and associate it with the abstract mileage value. In these listings, an `ObjectProperty` relates two instances of two classes, while a `DatatypeProperty` relates the instance of a class to a pre-defined type (integer, string, etc.).

Listing 1.6. `legacyUS.owl`:Ontology structure for the information in the relational database

```
<?xml version="1.0"?>
<rdf:RDF[...]
  xml:base="legacyUS.owl">
  <owl:Ontology rdf:about=""/>
  <owl:Class rdf:ID="Car"/>
  <owl:Class rdf:ID="Mileage"/>
  <owl:ObjectProperty rdf:ID="hasMileage">
    <rdfs:domain rdf:resource="#Car"/>
    <rdfs:range rdf:resource="#Mileage"/>
```

```
  </owl:ObjectProperty>
  <owl:DatatypeProperty rdf:ID="hasValue">
    <rdfs:range rdf:resource="[...]#float"/>
    <rdfs:domain rdf:resource="#Mileage"/>
  </owl:DatatypeProperty>
  <owl:FunctionalProperty rdf:ID="hasName">
    <rdfs:range rdf:resource="[...]#string"/>
    <rdfs:domain rdf:resource="#Car"/>
    <rdf:type rdf:resource="#DatatypeProperty"/>
  </owl:FunctionalProperty>
</rdf:RDF>
```

Listing 1.7. `legacyUSdata.owl`:Data represented using the ontological structure

```
<?xml version="1.0"?>
<rdf:RDF [...]
  xmlns:US="legacyUS.owl#"
  xmlns:contexts="contexts.owl#"
  xml:base="legacyUSdata.owl">
<owl:Ontology rdf:about="">
<owl:imports>
 <rdf:Description rdf:about="legacyUS.owl">
 </rdf:Description>
</owl:imports>
</owl:Ontology>
  <US:Mileage rdf:ID="mileage1">
    <US:hasValue [...]">24</US:hasValue>
  </US:Mileage>
  <US:Car rdf:ID="Focus">
  <US:hasName [...]>Ford Focus<US:hasName>
    <US:hasMileage rdf:resource="#mileage1"/>
  </US:Car>[...]
</rdf:RDF>
```

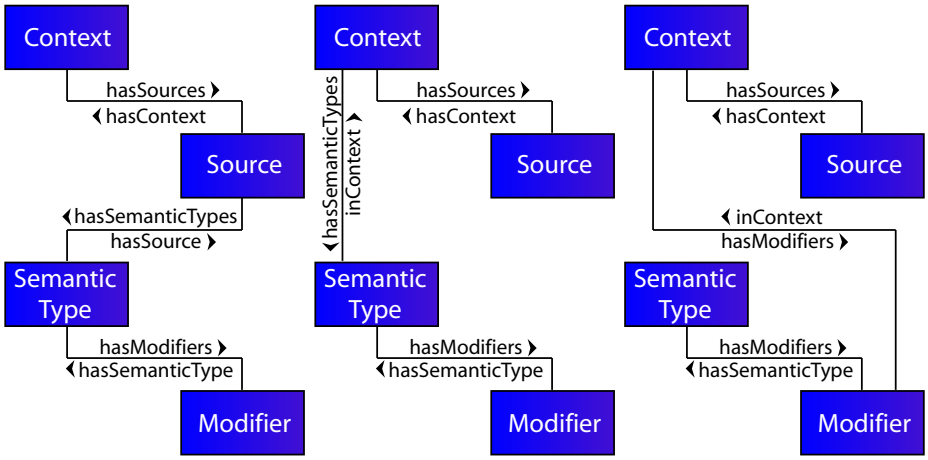
Listings 1.6 and 1.7 show the kind of input our system considers as *source*. We call the files *legacy* because they are obtained directly from existing data, without any context information. With respect to the amount of reasoning necessary at this point, our requirements are quite low since the machine needs not understand the concepts, but merely identify them as concepts rather than rows or columns in a table. The translation from the relational-model representation to our ontological representation is easily done automatically using one of the several available transformation languages such as XSLT [Cla99], FleXML [Ros01] or HaXml [WR99]. Subsequently, we can use expressions like `isSemanticType(Car, ST398)` to express the fact that the type `Car` defined in Listing 1.6 represents the conceptual type $ST398$. We refer to such expression

as *elevation axioms* because they elevate the class `Car` from its meaning as a collection of entities in a legacy database, to a conceptual level. Using such elevation axioms instead of attaching properties to the original class `Car` reduces the amount of work that the user needs to do by increasing the reusability of the code.

4 Separating context from data representation

In order to be able to do the things outlined in Section 1.2 we first need to establish a way to represent context. The flexibility of the RDF and OWL languages allow for such a variety of architectures to be defined, that one of the problems we faced was focusing on one in particular, one that provides, in our opinion, the best solution for future extensions.

Initially, we had reduced the possibilities to three models (Figure 2).



(a) Semantic Type is part of the Source

(b) Semantic Type is defined in the Context

(c) Semantic Type is independent of the Context

Fig. 2. Models for context expression

4.1 Model 1: Semantic Type as part of the Source

This model was our initial approach and it states that a data source should have a context and that it should contain a set of *semantic types*. It is the most basic approach because it attempts to link everything directly to the legacy data source.

This model was not eventually acceptable because a source should not actually contain a *semantic* object. It contains objects that we subsequently identify

as being automobiles, mileages, temperatures, locations, or anything else. By itself, it contains only some non-identifiable classes and objects that, in the COIN methodology, have to be related to semantic types via the elevation axioms.

4.2 Model 2: Semantic Type as part of the Context

From the first model, we have learned that the semantic types need to be defined separately from the data itself. Consequently, we considered having them defined as part of the context. This method provides sufficient flexibility to allow each user that defines his or her own context to have complete freedom as to what it considers to be significant types and how these should be represented.

The disadvantage of the method also lies in the flexibility we just mentioned: additional mediation is needed and even if two users define two contexts with semantic types having exactly the same representation, they still appear as duplicates when everything is put together to allow query answering. (see Figure 3: the instance browser at the middle of the image shows duplicate semantic types corresponding to each of the two contexts defined)

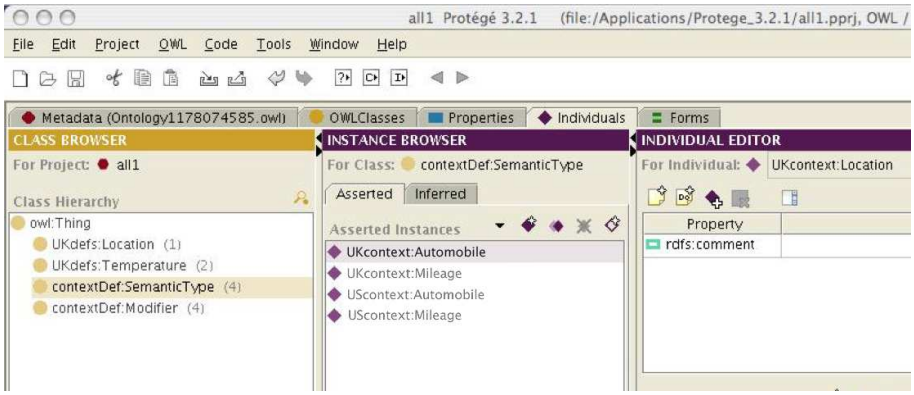


Fig. 3. Duplicate definitions of Semantic types using the second model

4.3 Model 3: Semantic types defined independently of everything else

Finally, the chosen model considers the semantic types to be independent of both the context and the source. In fact, we should imagine these semantic types as defined in an external ontology. This method provides the most independence between the different concepts. Figure 4 presents a more detailed view than the one in Figure 2, showing the files used for each component and referencing Listings presented in this paper. We will be using three files to express context, in addition to the two files that represent the data. Two of the three are also

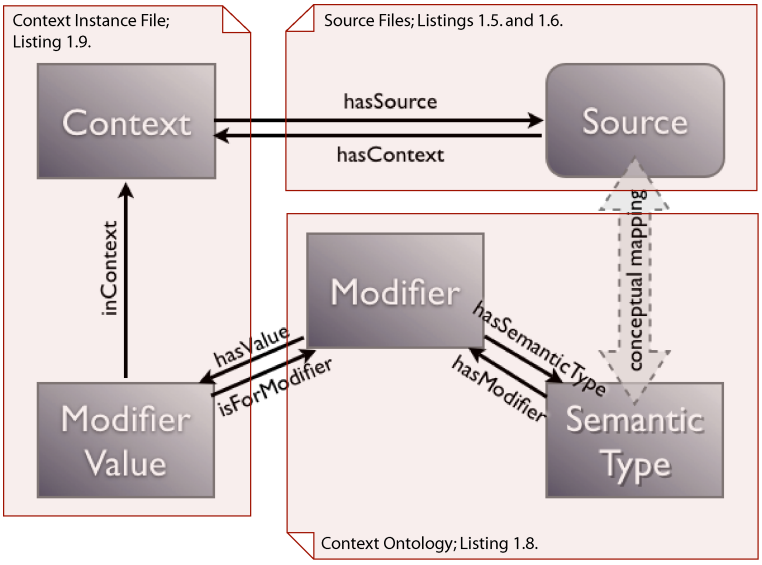


Fig. 4. Our model, along with the files that contain each component

represented in Figure 4, while the third - the `contextDefs.owl` file in Listing 1.8 provides the basic definitions needed for context representation. As such, it can be though of as the entire Figure 4 (without the sources). The `contextDefs.owl` files defines the context as a set of modifiers attached to a semantic type (some items present in the file will be explained in the next sections).

Listing 1.8. `contextDefs.owl`: the context definition

```
<?xml version="1.0"?>
<rdf:RDF
  xml:base="contextDefs.owl">
  <owl:Ontology rdf:about=""/>
  <owl:Class rdf:ID="Context"/>
  <owl:Class rdf:ID="Query"/>
  <owl:Class rdf:ID="TriggeredRules"/>
  <owl:Class rdf:ID="SemanticType"/>
  <owl:Class rdf:ID="Modifier"/>
  <owl:Class rdf:ID="ModifierValue"/>
  <owl:ObjectProperty rdf:ID="hasModifiers">
    <rdfs:range rdf:resource="#Modifier"/>
    <rdfs:domain rdf:resource="#SemanticType"/>
  </owl:ObjectProperty>
  <owl:ObjectProperty rdf:ID="isSemanticType">
    <rdfs:range rdf:resource="#SemanticType"/>
  </owl:ObjectProperty>
```

```
18    <owl:DatatypeProperty rdf:ID="hasValue">
19      <rdfs:range rdf:resource="[...]#string"/>
20      <rdfs:domain rdf:resource="#ModifierValue"/>
21    </owl:DatatypeProperty>
22    <owl:ObjectProperty rdf:ID="hasContext"/>
23     <owl:ObjectProperty rdf:ID="inContext">
24      <rdfs:range rdf:resource="#Context"/>
25      <rdfs:domain rdf:resource="#ModifierValue"/>
26    </owl:ObjectProperty>
27     <owl:ObjectProperty rdf:ID="isForModifier">
28      <rdfs:range rdf:resource="#Modifier"/>
29    </owl:ObjectProperty>
30      <owl:DatatypeProperty rdf:ID="ruleName">
31      <rdfs:domain rdf:resource="#TriggeredRules"/>
32      <rdfs:range rdf:resource="[...]#string"/>
33    </owl:DatatypeProperty>[...]
34  </rdf:RDF>
```

The following listing (Listing 1.9) contains the file that represents the system's *understanding of the world*: a list of concepts, what properties they have (modifiers) and how they relate to each other. In our example, it states that automobiles have a *mileage* attribute that is measured by *mileage unit*.

Listing 1.9. `contextOntology.owl`: Semantic types definitions (including the modifiers they accept)

```
1   <?xml version="1.0"?>
2   <rdf:RDF
3    xmlns="contextOntology.owl#"
4    xmlns:contextDef="contextDefs.owl#"
5    xml:base="contextOntology.owl">
6    <owl:Ontology rdf:about="">
7     <owl:imports rdf:resource="contextDefs.owl"/>
8     </owl:Ontology>
9     <contextDef:SemanticType rdf:ID="Mileage">
10      <contextDef:hasModifiers>
11       <contextDef:Modifier rdf:ID="MileageUnit"/>
12      </contextDef:hasModifiers>
13    </contextDef:SemanticType>
14    <contextDef:SemanticType rdf:ID="Automobile"/>
15  </rdf:RDF>
```

Finally, Listing 1.10 shows the instance of a context: all, or just a subset of modifiers, are assigned values in this file. In this listing, Lines 10-13 give a label to the context, which will be used in differentiating modifier values with similar representations in different contexts (like *mpg* in our case). Then, lines 14-20

define the *mpg* modifier value, identifying its context and the modifier it applies to.

Listing 1.10. `UScontext.owl`: Context instance file

```
1  <?xml version="1.0"?>
2  <rdf:RDF
3   xmlns="UScontext.owl#"
4   xmlns:contextOntology="contextOntology.owl#"
5   xmlns:contextDef="contextDefs.owl#"
6   xml:base="UScontext.owl">
7   <owl:Ontology rdf:about="">
8    <owl:imports rdf:resource="contextOntology.owl"/>
9   </owl:Ontology>
10  <contextDefs:Context rdf:id="USContext">
11    <rdfs:label rdf:datatype="[...]#string">
12      USContext</rdfs:label}
13  </contextDefs:Context>
14  <contextDefs:ModifierValue rdf:ID="mpg">
15   <contextDefs:inContext rdf:resource="#USContext"/>
16   <contextDefs:isForModifier rdf:resource=
17    "contextOntology.owl#MileageUnit"/>
18   <contextDefs:hasValue rdf:datatype="[...]#string">
19      mpg</contextDefs:hasValue>
20   </contextDefs:ModifierValue>
21 </rdf:RDF>
```

4.4 Automatic identification of Context

In Listing 1.10 we identified explicitly the context of a modifier value by means of the `inContext` property in line 15. This may seem redundant considering that the value is defined in a file specific to this context. In fact, this property can be automatically asserted using the `from named` construct in the SPARQL query language. Listing 1.11 shows how this can be done, with results in Table 2.

Listing 1.11. Using the `from named` construct we can identify to which context each modifier value belongs to

```
1  prefix contextDefs:<contextDefs.owl#>
2  select ?src ?modifier ?value
3  from named <UKcontext.owl>
4  from named <UScontext.owl>
5  where {
6   graph ?src{
7    ?modifier contextDefs:hasValue ?value
8   }
9  }
```

Table 2. Result of query in listing 1.11

src	modifer	value
contextUS.owl	contextOntology:MileageUnit	mpg
contextUK.owl	contextOntology:MileageUnit	mpg

The results in column *src* of Table 2 may be used to replace the labels in Listing 1.10. In the remaining of the presentation we will assume that the labels have already been set, either manually, or using the method we just indicated.

4.5 The mediator file

To make the system work, one file needs to import all these bits and pieces together and build the construct of the COntext INtegration strategy. We call this the *mediator* file. Listing 1.12 shows an extract of its contents, in particular the `import` statements and the way we define the context of a source. In this example, the source is just the Mileage entity `mileage1` defined in the `legacyUSdata.owl` file (Listing 1.7). However, it can be anything else: an entire file, a class or just an instance as in this case.

Listing 1.12. The *mediator* file puts together all the different pieces of the architecture and defines particular contexts of the sources

```
1  <rdf:RDF>
2   xmlns:UKdefs="legacyUK.owl#"
3   xmlns:USdefs="legacyUS.owl#"
4   xmlns:USdata="legacyUSdata.owl#"
5   xmlns:contextDefs="contextDefs#"
6   [...]
7   <owl:Ontology rdf:about="">
8     <owl:imports rdf:resource="UScontext.owl"/>
9     <owl:imports rdf:resource="UKcontext.owl"/>
10    <owl:imports rdf:resource="legacyUSdata.owl"/>
11    <owl:imports rdf:resource="legacyUK.owl"/>
12  </owl:Ontology>
13  <rdf:Descripton rdf:about="legacyUSdata.owl#mileage1">
14    <j.0:isSemanticType rdf:resource=
15    "contextOntology.owl#Mileage"/>
16    <j.0:hascontext rdf:resource="UScontext.owl#"/>
17   </rdf:Description>
18 </rdf:RDF>
```

In the listing above, lines 2-4 give names to particular ontologies, names which we will use in defining the rules and the queries below.

Now we can, for instance, identify the context of a source using a query similar to the following:

Listing 1.13. Query to identify the context of a source

```
prefix contextDefs:<contextDefs.owl#>
select ?data ?context
 where {?data contextDefs:hasContext ?context}
```

In our example, the results of this query is shown in Table 3.

Table 3. Results of the context query in Listing 1.13

data	context
USdata:mileage1	USContext:USContext

5 Context conflict identification and resolution

In the previous sections we have explained how a user might query the data to find out what is the appropriate context it refers to. In this section we will show how we do this automatically for the purpose of context conflict determination and how this determination will trigger the necessary conversion rules. We will continue to use the example of cars and mileages presented throughout this work.

The approach we follow in this work is a two-step approach: first, we need to determine the need for a conversion (i.e. determine the existence of two different contexts) and then apply the corresponding rule.

These two phases are implemented in two sets of rules: first a *trigger rule* analyses the data and the query to identify potential conflicts. If one such conflict is identified, a flag is raised, to announce the necessity of the application of a conversion rule. We will describe the implementation of this flag shortly. Upon assertion of the trigger flag, the corresponding rule will automatically be triggered and context mediation will take place by addition of new data to the dataset.

5.1 The trigger rule

The idea of the trigger rule is to look for conflicts and add a flag, in the form of a small text representing the needed conversion, which is added to a collection of triggers called `TriggeredRules1` in our example.

Listing 1.14 shows the exact rule used to determine conflicts between any two attributes of the same type.

Listing 1.14. Rule for the determination of context conflict

```
USdefs:hasValue(?attribute, ?attributeValue) ∧
contextDefs:hascontext(?attribute, ?dataContext) ∧
contextDefs:hascontext(Query_1, ?queryContext) ∧
differentFrom(?dataContext, ?queryContext) ∧
```

```
5  contextDefs:isSemanticType(?temp, ?semType) ∧
6  contextDefs:hasModifiers(?semType, ?modifier) ∧
7  contextDefs:isForModifier(?modVal, ?modifier) ∧
8  contextDefs:hasValue(?modVal, ?dataModVal) ∧
9  contextDefs:inContext(?modVal, ?datacontext) ∧
10 contextDefs:isForModifier(?modVal1, ?modifier) ∧
11 contextDefs:hasValue(?modVal1, ?queryModVal) ∧
12 contextDefs:inContext(?modVal1, ?querycontext) ∧
13 rdfs:label(?queryContext, ?c1) ∧
14 rdfs:label(?dataContext, ?c2) ∧
15 swrlb:stringConcat(?tn0,":",?queryModifierValue) ∧
16 swrlb:stringConcat(?tn1,?c1,?tn0) ∧
17 swrlb:stringConcat(?tn2,"-to-",?tn1) ∧
18 swrlb:stringConcat(?tn3,?dataModifierValue,?tn2) ∧
19 swrlb:stringConcat(?tn4,":",?tn3) ∧
20 swrlb:stringConcat(?triggername,?c2,?tn4)
21  → contextDefs:ruleName(TriggeredRules1, ?triggername)
```

A detailed explanation follows:

Lines 1-4 identify the difference between the contexts of the data and the query.
Lines 5-6 identify the semantic type and modifier
Lines 7-9 identify the value of the modifier in the context of the dataset
Lines 10-12 do the same for the value of the modifier in the query context.
Lines 13-14 identify the labels of each context, used later in generating the trigger name
Lines 15-20 generate the name of the trigger
Line 21 asserts the trigger

In this implementation, the attribute itself is linked by a `hasContext` relation to a particular context. In other situations, such a relation may only be defined for the entire dataset, rather than for individual attributes. This is not a problem, as a SWRL rule can extend the `hasContext` rule from a class to its components.

In our running example, this rule would generate a flag of the form *USContext:mpg-to-UKContext:mpg*.

5.2 The conversion rule

The actual conversion rule that transforms the miles per gallon measure unit from Winchester gallons to Imperial gallons is shown in Listing 1.15. Line 1 checks the existence of the triggered flag and, if this condition is satisfied, it performs the necessary mathematical conversion functions (lines 2-4) and asserts the new value in line 5.

The result of applying both rules on the knowledge base is shown in Figure 5. We can see that two new facts have been asserted: first, the trigger rule has discovered the context conflict and, second, upon assertion of the conflict, the conversion rule has been triggered to compute the new value.

Listing 1.15. Conversion rule

```
contextDefs:ruleName(TriggeredRules1,
        "USContext:mpg-to-UKContext:mpg") ∧
USdefs:hasValue(?mileage, ?mileageValue) ∧
swrlb:multiply(?mileage1, ?mileageValue, 12) ∧
swrlb:divide(?newValue, ?mileage1, 10)
  →  UKdefs:hasValue(?mileage, ?newValue)
```

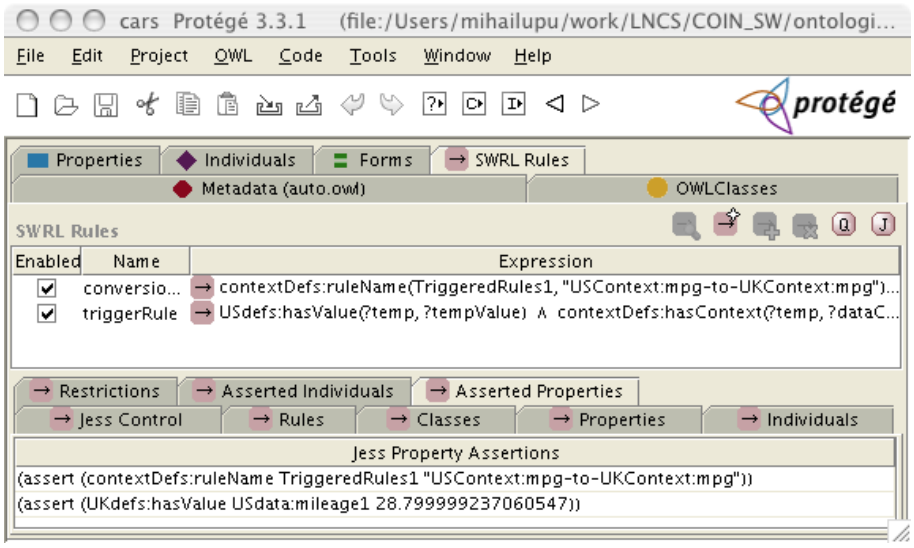


Fig. 5. With two rules in the knowledge base, the conversion between mileage units has been performed automatically. In the lower half of the image we can see that the new mileage value (28.8) has been correctly asserted

5.3 Query alteration

The simple way in which we have defined the conversion rule (Listing 1.15) allows us to re-write the query in an equally simple manner. Listings 1.16 and 1.17 show the original and, respectively, the new query for obtaining the value of the mileage for the Focus. As it can be observed, the only difference is the prefix of the `hasValue` relationship.

Listing 1.16. Original query

```
SELECT ?mileageValue
WHERE { ?loc USdefs:hasMileage ?mileage.
              ?loc USdefs:hasName "Ford Focus".
              ?mileage USdefs:hasValue ?mileageValue}
```

Listing 1.17. New query for the UK context

```
SELECT ?mileageValue
WHERE { ?loc USdefs:hasMileage ?mileage.
                ?loc USdefs:hasName "Ford Focus".
                ?mileage UKdefs:hasValue ?mileageValue}
```

Despite its ease of use and implementation, the current method is not perfect. The relation `UKdefs:hasValue` is not directly linked to the UK context. Formally, it has no link to any specific context. Though this approach can be implemented programatically, our future work, described in the next section, aims towards an ever closer integration with the semantic web tools.

6 Future work

Our work so far has shown how we can approach the problem of context interchange using the COIN strategy via the tools of the Semantic web. To fully achieve all the features that are currently available in COIN there are still steps ahead, some of which we describe in this section.

The solution presented in the previous section relies on external programing languages to transform a query such that it returns the result in a different context. A better solution would be to have a new tertiary relation, similar to the one that defined the value of a modifier in a particular context. This new relation, which we call `hascontextValue` links together an attribute, a value and a context. As we have seen, SWRL can only express binary relations directly, so the only way to implement this relation is to define it as an owl:class with three binary relations. Now, the conversion rule needs to infer the new tertiary relation that links the attribute to the new value in the new context. Such a rule can be created following the Semantic Web best practices [NRHW06] as in Listing 1.18.

Listing 1.18. Tertiary relation implemented as an OWL class

```
<owl:Class rdf:ID="hascontextValueRelation"/>
  <owl:ObjectProperty
     rdf:ID="hascontextValueRelation_context">
    <rdfs:range rdf:resource="#Context"/>
  </owl:ObjectProperty>
  <owl:ObjectProperty
     rdf:ID="hascontextValueRelation_attribute"/>
  <owl:ObjectProperty
     rdf:ID="hascontextValueRelation_value"/>
```

The difficulty in inferring this relation in the conversion rule is that a new instance has to be generated: a new individual of the `hascontextValue` type that would link the three components (attribute, value and context). Unfortunately, the current standard SWRL specification does not provide means to instantiate

classes, thus making this solution temporarily unfeasible. This leavs only the option of an "impure" approach using external programming tools.

7 Conclusion

In this work we describe how the COntext INterchange strategy can be implemented using the Semantic Web tools, in particular using OWL, SWRL and SPARQL. We acknowledge the existence of massive amounts of data in relational databases that lack all the necessary data required for users other than the original designers of the database and describe how the information present in these databases can be "elevated" to a knowledge base. Subsequently, we show how to structure information pertaining to the context of the data - how to model the definitions of *semantic type*, *modifier* and *modifier value*. Using these models we show how the necessary conversions of the data values can be made by using a two-step process involving pairs of *trigger* and *conversion* rules.

References

[Ast04] I. Astrova. *The Semantic Web: Research and Applications*, chapter Reverse Engineering of Relational Databases to Ontologies. Springer Berlin / Heidelberg, 2004.

[BLHL01] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, 2001.

[bos] Bossam. http://bossam.wordpress.com/.

[Cla99] J. Clark. Xsl transformations. http://www.w3.org/TR/xslt, 1999.

[Fir03] A. Firat. *Information Integration using Contextual Knowledge and Ontology Merging*. PhD thesis, Sloan Business School, MIT, 2003.

[GBMS99] C. H. Goh, S. Bressan, S. Madnick, and M. Siegel. Context interchange: New features and formalisms for the intelligent integration of information. *ACM TIS*, 17(3), 1999.

[hoo] Hoolet. http://owl.man.ac.uk/hoolet/.

[HPSB+04] I. Horrocks, P. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. SWRL: A semantic web rule language combining OWL and RuleML. http://www.w3.org/Submission/SWRL/, 2004.

[jes] The Jess inference engine. http://herzberg.ca.sandia.gov/jess/.

[kao] Kaon2. http://kaon2.semanticweb.org/.

[LM04] N. Lammari and E. Metais. Building and maintaining ontologies: a set of algorithms. *NLDB*, 48(2), 2004.

[LM07] M. Lupu and S. Madnick. Using semantic web tools for context interchange. In *Proc. of SWDB-ODBIS Workshop*, 2007.

[Mad01] S. Madnick. The misguided silver bullet: What XML will and will not do to help information integration. In *Procs. of the iiWAS*, 2001.

[Mel06] J. Melton. SQL, XQuery, and SPARQL:what's wrong with this picture? In *Proc. of XTech Conference*, 2006.

[MvH04] D. McGuinness and F. van Harmelen. Owl web ontology language overview. http://www.w3.org/TR/owl-features/, 2004.

[NAS] NASA. Mars climate orbiter failure causes. http://mars.jpl.nasa.gov/msp98/news/mco990930.html.

[NRHW06] N. Noy, A. Rector, P. Hayes, and C. Welty. Defining n-ary relations on the semantic web. http://www.w3.org/TR/swbp-n-aryRelations/, 2006.

[pel] Pellet. http://www.mindswap.org/2003/pellet/.

[pot] Potluck mash-up tool. http://dfhuynh.csail.mit.edu:6666/potluck/.

[pro] Protégé. http://protege.stanford.edu/.

[PS07] E. Prud'hommeaux and A. Seaborne. Sparql query language for rdf. http://www.w3.org/TR/rdf-sparql-query/, 2007.

[r2m] R2ml. http://oxygen.informatik.tu-cottbus.de/rewerse-i1/?q=node/6.

[rac] RacerPro. http://www.racer-systems.com/products/racerpro/index.phtml.

[Ros01] Kristoffer Rose. FleXML - XML Processor Generator. http://flexml.sourceforge.net, 2001.

[swr] SWRLTab. http://protege.cim3.net/cgi-bin/wiki.pl?SWRLTab.

[WR99] M. Wallace and C. Runciman. Haskell and XML: Generic Combinators or Type-Based Translation? In *Proc. of the International Conference on Functional Programming*. http://www.cs.york.ac.uk/fp/HaXml/, 1999.