p 123

Using Semantic Web tools for COntext INterchange

Mihai Lupu
Singapore-MIT Alliance
National University of Singapore
mihailup@comp.nus.edu.sg

Stuart Madnick
Sloan School of Management
Massachusetts Institute of Technology
smadnick@mit.edu

ABSTRACT

The COntext INterchange Strategy (COIN) 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, 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 problems 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 [11] 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, outsourcing 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[1] project developed at the Computer Science and AI Laboratory at MIT, 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 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.

Returning to our *Weather* example used in the previous section, Listing 1 shows how we might represent different values of the same temperature using prefixes of relations.

Listing 1: A possible solution to representing context, by adding an additional *hasValue* relation to the Temperature object. It is simple and intuitive of the fact that we are dealing indeed with the same temperature.

```
<?xml version="1.0"?>
<rdf:RDF
```

[1] http://dfhuynh.csail.mit.edu:6666/potluck/

VLDB '07, September 23-28, 2007, Vienna, Austria.

```
<UK:Temperature rdf:ID="temp1">
  <UK:hasValue rdf:datatype="[...]#int"
  >10</UK:hasValue>
  <USA:hasValue rdf:datatype=[...]#int>
  50
  </USA:hasValue>
</UK:Temperature>
<UK:Location rdf:ID="London">
  <UK:hasName rdf:datatype="[...]#string">
    London, UK</UK:hasName>
  <UK:hasTemperature rdf:resource="#temp1" />
</UK:Location>
</rdf:RDF>
```

The way to add this new relation is by defining a SWRL [5] rule such as the one in Listing 2 and then querying the results with a query in SPARQL [13] as in Listing 3. This all seems very intuitive. As always, the problems lie in the details: How do we detect a conflict of contexts? How do we identify the correct rule to be applied? Where is this rule stored? 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?

We will answer all these questions in the following sections, starting in Section 4 with the basic representation of context using the Ontology Web Language (OWL [9]). For a more detailed discussion, we refer the reader to [7]

Listing 2: SWRL rule that generates a value in the USA context

```
UK:hasValue(?temp, ?tempValue) ^
UK:hasTemperature(?location, ?temp) ^
swrlb:multiply(?product,?tempValue,1.8) ^
swrlb:add(?sum, ?product, 32) ->
  USA:hasValue(?temp, ?sum)
```

Listing 3: SPARQL query that retrieves the value in the US context

```
SELECT ?tempValue
WHERE { ?l UK:hasTemperature ?temp.
        ?l UK:hasName "London" .
        ?temp USA:hasValue ?tempValue
      }
```

2. BACKGROUND AND RELATED WORK

2.1 COntext INterchange

The idea of the COntext INterchange [4] system is to reuse massive amounts of data that already exist but that are incomplete due to design assumptions that eliminated constants from the dataset. When two or more such datasets are put together, or queried together, what used to be 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 *Weather* example of Section 3, the column `temperature` has no meaning by itself until it is associated with a semantic type *Temperature*. 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 `temp` 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, *Temperature* has only one attribute, *value*.

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

COIN uses this architecture to automatically detect differences in contexts and resolve queries in the context of 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. [3].

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. [2] 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) [9]. 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 [13]. 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 [10], 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.

Last but not least, 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 [5] 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[2] - an extension to the Protégé framework, that uses mainly the

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

Jess[3] inference engine (though it could use other engines too).

3. FROM TABLES TO INFORMATION

The first step towards making the data understandable by different agents[4] 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. This can be easily done, as most database systems have the capacity to dump the data in this format. For instance, a simple table containing locations and temperatures (Table 1) can be expressed as in Listing 4.

However, using XML does not solve our problem. As discussed in [8], XML is not a silver bullet - it is just another way to express the data. It only provides a more flexible way to express the data, 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 4, 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.

Location	Temperature
london	10
paris	12

Table 1: Sample relational table

Listing 4: XML representation of relational database

```
<?xml version="1.0"?>
<mysqldump xmlns:xsi="[...]XMLSchema-instance">
<database name="test">
 <table_structure name="weatherUK">
  <field Field="location" Type="varchar(20)" />
  <field Field="temperature" Type="int(11)" />
 </table_structure>
  <table_data name="weatherUK">
   <row>
    <field name="location">london</field>
    <field name="temperature">10</field>
   </row>
[...]
  </table_data>
 </database>
</mysqldump>
```

There exist attempts to extract ontological information from relational tables [1, 6]. What we want here is not nearly as ambitious as the works mentioned before. We don't necessarily want here to infer an 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 5 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 (*location* and *temperature*) who are connected by a relationship *hasTemperature*. The difference between this approach and the simple XML dump is that here *location* and *temperature* are regarded as concepts, rather than fields in a table. It is a subtle, but essential difference. Here, a particular instance of a Location has a name, but is separate from its name. This distinction will allow us later to specify that *London* is the same location with *Londres* and that 10 is the same temperature with 50 (one in Celsius and one in Fahrenheit degrees). This way, in the data file shown in Listing 6 we can define the abstract temperature *temp1* and give it a value and then define the abstract location *London* and give it a name and associate it with a temperature value.

Listing 5: Ontology structure for the information in the relational database

```
<?xml version="1.0"?>
<rdf:RDF [...]
   xml:base="legacyUK.owl">
  <owl:Ontology rdf:about=""/>
  <owl:Class rdf:ID="Location"/>
  <owl:Class rdf:ID="Temperature"/>
  <owl:ObjectProperty rdf:ID="hasTemperature">
    <rdfs:domain rdf:resource="#Location"/>
    <rdfs:range rdf:resource="#Temperature"/>
  </owl:ObjectProperty>
  <owl:DatatypeProperty rdf:ID="hasValue">
    <rdfs:range rdf:resource="[...]#int"/>
    <rdfs:domain rdf:resource="#Temperature"/>
  </owl:DatatypeProperty>
  <owl:FunctionalProperty rdf:ID="hasName">
    <rdfs:range rdf:resource="[...]#string"/>
    <rdfs:domain rdf:resource="#Location"/>
    <rdf:type rdf:resource="#DatatypeProperty"/>
  </owl:FunctionalProperty>
</rdf:RDF>
```

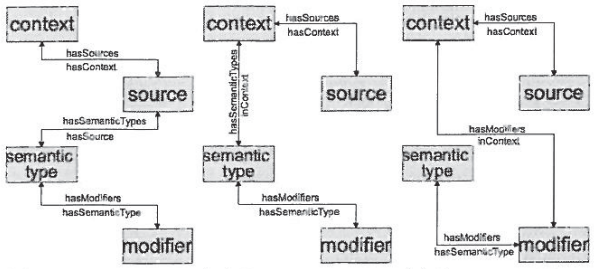
Listing 6: Data represented using the ontological structure

```
<?xml version="1.0"?>
<rdf:RDF [...]
   xmlns:UK="legacyUK.owl#"
   xmlns:cntxts="cntxts.owl#"
   xml:base="legacyUKdata.owl">
<owl:Ontology rdf:about="">
<owl:imports>
 <rdf:Description rdf:about="legacyUK.owl">
 </rdf:Description>
</owl:imports>
</owl:Ontology>
  <UK:Temperature rdf:ID="temp1">
    <UK:hasValue [...]">10</UK:hasValue>
  <UK:Temperature>
  <UK:Location rdf:ID="London">
  <UK:hasName [...]>London, UK<UK:hasName>
    <UK:hasTemperature rdf:resource="#temp1"/>
  <UK:Location>[...]
</rdf:RDF>
```

In this section we have shown how data must be represented such that automated context mediation may be possible. Our requirements are in fact quite low with respect to the amount of reasoning necessary at this point. The translation from the relational-model representation to our

[3] http://herzberg.ca.sandia.gov/jess/

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



(a) Semantic Type is part of the Source (b) Semantic Type is defined in the Context (c) Semantic Type is independent of the Context

Figure 1: Models for context expression

ontological representation is easily done automatically, as we do not require the machine to understand the concepts, but merely to identify them as concepts rather than rows or columns in a table.

4. SEPARATING CONTEXT FROM DATA REPRESENTATION

In order to be able to do the things outlined in Section 1.1 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, depicted in Figure 1.

Model 1: Semantic Type as part of the Source. This model was our initial approach and it states that a source has a context and that it contains a set of *semantic types*. In our example from Section 3, the semantic types would be *Temperature* or *Location*.

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 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.

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.

Model 3: Semantic types defined independently of everything else. Finally, the third 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. We will be using three files to express context, in addition to the two files that represent the data (the ontology and the instances).

First, Listing 7 defines what a context is: a set of modifiers attached to a semantic type (some items present in the file will be explained in the next sections).

Listing 7: Context definition

```
<?xml version="1.0"?>
<rdf:RDF
  xmlns:p1="cntxtDefs.owl#"
  xml:base="cntxtDefs.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>
  <owl:DatatypeProperty rdf:ID="hasValue">
    <rdfs:range rdf:resource="[...]#string"/>
    <rdfs:domain rdf:resource="#Modifier"/>
  </owl:DatatypeProperty>
  <owl:ObjectProperty rdf:ID="hasCntxt"/>
   <owl:ObjectProperty rdf:ID="inContext">
    <rdfs:range rdf:resource="#Context"/>
    <rdfs:domain rdf:resource="#ModifierValue"/>
  </owl:ObjectProperty>
   <owl:ObjectProperty rdf:ID="isForModifier">
    <rdfs:range rdf:resource="#Modifier"/>
  </owl:ObjectProperty>
    <owl:DatatypeProperty rdf:ID="ruleName">
    <rdfs:domain rdf:resource="#TriggeredRules"/>
    <rdfs:range rdf:resource="[...]#string"/>
  </owl:DatatypeProperty>[...]
</rdf:RDF>
```

The following listing (Listing 8) 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.

Listing 8: This file contains the semantic types and the kinds of modifiers they accept

```
<?xml version="1.0"?>
<rdf:RDF
 xmlns="cntxtOntology.owl#"
 xmlns:cntxtDef="cntxtDefs.owl#"
 xml:base="cntxtOntology.owl">
 <owl:Ontology rdf:about="">
  <owl:imports rdf:resource="cntxtDefs.owl"/>
  </owl:Ontology>
  <cntxtDef:SemanticType rdf:ID="Temperature">
    <cntxtDef:hasModifiers>
     <cntxtDef:Modifier rdf:ID="TemperatureUnit"/>
    </cntxtDef:hasModifiers>
  </cntxtDef:SemanticType>
  <cntxtDef:SemanticType rdf:ID="Location"/>
</rdf:RDF>
```

Finally, Listing 9 shows the instance of a context: all, or just a subset of modifiers, are assigned values in this file.

Listing 9: Context instance file

```
<?xml version="1.0"?>
<rdf:RDF
 xmlns="UKcntxt.owl#"
 xmlns:cntxtOntology="cntxtOntology.owl#"
 xmlns:cntxtDef="cntxtDefs.owl#"
```

```
 xml:base="UKcntxt.owl">
 <owl:Ontology rdf:about="">
  <owl:imports rdf:resource="cntxtOntology.owl" />
 </owl:Ontology>
 <cntxtDefs:ModifierValue rdf:ID="C">
 <cntxtDefs:inContext rdf:resource="#UKContext" />
 <cntxtDefs:isForModifier rdf:resource=
  "cntxtOntology.owl#TemperatureUnit" />
 <cntxtDefs:hasValue rdf:datatype="[...]# string"
    >C</cntxtDefs:hasValue>
  </cntxtDefs:ModifierValue>
</rdf:RDF>
```

In this model a basic query cannot differentiate between the two values of a modifier, even if they were defined in different files. This issue can be easily overcome using the `from named` construct in the SPARQL query language, as we show in [7].

4.1 The mediator file

Finally, 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 10 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 Temperature entity `temp1` defined in the `UKInstances.owl` file. However, it can be anything else: an entire file, a class or just an instance as in this case.

Listing 10: The *moderator* file puts together all the different pieces of the architecture and defines particular contexts of the sources

```
  <owl:Ontology rdf:about="">
    <owl:imports rdf:resource="USAcntxt.owl" />
    <owl:imports rdf:resource="UKcntxt.owl" />
    <owl:imports rdf:resource="UKinstances.owl" />
  </owl:Ontology>
<rdf:Descripton rdf:about="UKinstances.owl#temp1">
    <j.0:isSemanticType rdf:resource=
    "cntxtOntology.owl#Temperature" />
    <j.0:hasCntxt rdf:resource="UKcntxt.owl#" />
  </rdf:Description>
```

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

Listing 11: Query to identify the context of a source

```
prefix cntxtDefs:<cntxtDefs.owl#>
prefix mediator:<all.owl#>
select ?data ?context
from <all.owl>
 where {?data cntxtDefs:hasCntxt ?context}
```

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 if refers to. In this section we will show how we do this automatically for the purpose of context conflict detection and how this detection will trigger the necessary conversion rules. We will continue to use the example of locations and temperatures present throughout this work.

The approach we follow in this work is a two-step approach: first, we need to detect the need for a transformation (i.e. detect 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 transformation 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.

The trigger rule. Listing 12 shows the exact rule used to detect conflicts between any two attributes of the same type. Lines 1-4 identify the difference between the contexts of the data and the query. In this implementation, the attribute itself is linked by a `hasCntxt` 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 `hasCntxt` rule from a class to its components.

Listing 12: Context conflict detection rule

```
UKdefs:hasValue(?attribute, ?attributeValue)^
cntxtDefs:hasCntxt(?attribute, ?dataContext)^
cntxtDefs:hasCntxt(Query_1, ?queryContext)^
differentFrom(?dataContext, ?queryContext)^
cntxtDefs:isSemanticType(?temp, ?semType)^
cntxtDefs:hasModifiers(?semType, ?modifier)^
cntxtDefs:isForModifier(?modVal, ?modifier)^
cntxtDefs:hasValue(?modVal, ?dataModVal)^
cntxtDefs:inContext(?modVal, ?dataCntxt)^
cntxtDefs:isForModifier(?modVal1, ?modifier)^
cntxtDefs:hasValue(?modVal1, ?queryModVal)^
cntxtDefs:inContext(?modVal1, ?queryCntxt)^
differentFrom(?dataModVal,?queryModVal)^
swrlb:stringConcat(?trigger1,"-to-",?queryModVal)^
swrlb:stringConcat(?trigger,?dataModVal,
?queryModVal)
 -> cntxtDefs:ruleName(TriggeredRules1, ?trigger)
```

The conversion rule. The actual conversion rule that transforms degrees in Celsius to degrees in Fahrenheit is shown in Listing 13. The result of applying both rules on the knowledge base is shown in Figure 2. 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 13: Conversion rule

```
cntxtDefs:ruleName(TriggeredRules1, "C-to-F")   ^
UKdefs:hasValue(?temp, ?tempValue)   ^
swrlb:multiply(?temp1, ?tempValue, 9)   ^
swrlb:divide(?temp2, ?temp1, 5)   ^
swrlb:add(?newValue, ?temp2, 32)
 ->  USAdefs:hasValue(?temp, ?newValue)
```

Query alteration. The simple way in which we have defined the conversion rule (Listing 13) allows us to re-write the query in an equally simple manner. Listings 14 and 15 show the original and, respectively, the new query for obtaining the value of temperature in London.

Listing 14: Original query

```
SELECT ?tempValue
WHERE { ?loc UKdefs:hasTemperature ?temp.
             ?loc UKdefs:hasName "London" .
             ?temp UKdefs:hasValue ?tempValue}
```

Listing 15: New query for the USA context

```
SELECT ?tempValue
WHERE { ?loc UKdefs:hasTemperature ?temp.
             ?loc UKdefs:hasName "London" .
             ?temp USAdefs:hasValue ?tempValue}
```

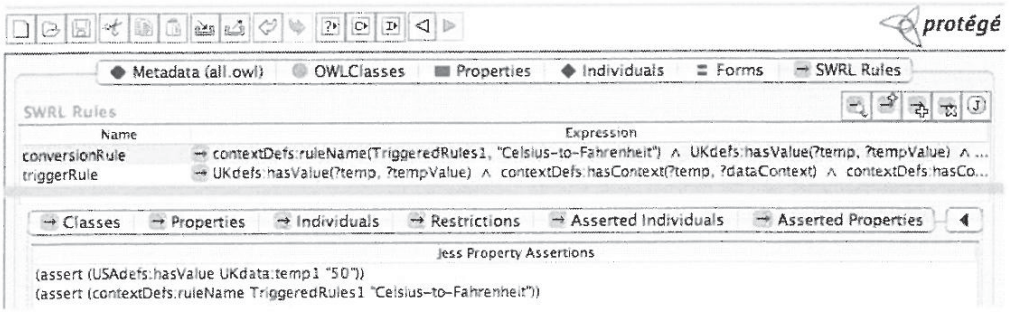


Figure 2: With two rules in the knowledge base, the conversion between temperature units has been performed automatically. In the lower half of the image we can see that the new temperature value (50F) has been correctly asserted

Despite it's ease of use and implementation, the current method is not perfect. The relation `USAdefs:hasValue` is not directly linked to the USA context. Formally, it has no link to any specific context. Though this approach can be implemented programatically our future work, as 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 `hasCntxtValue` 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. In [7] we have showed how such a rule can be created following the Semantic Web best practices [12].

The difficulty in inferring this relation in the conversion rule is that a new instance has to be generated: a new individual of the `hasCntxtValue` 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.

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.

8. REFERENCES

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

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

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

[4] 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.

[5] 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.

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

[7] M. Lupu and S. Madnick. Using semantic tools for context interchange. http://www.comp.nus.edu.sg/g0404405/coinsw.WP.pdf, 2007.

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

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

[10] J. Melton. SQL, XQuery, and SPARQL:whats wrong with this picture? In *Proc. of XTech Conference*, 2006.

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

[12] 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.

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

Table of Contents

See p. 43

Proceedings

SWDB-ODBIS07
Joint ODBIS & SWDB Workshop on Semantic Web, Ontologies, Databases

Colocated with VLDB2007, Vienna, September 24, 2007