Knowledge Conceptualization and Software Agent based Approach for OWL Modeling Issues

S. Zhao[1], P. Wongthongtham[2], E. Chang[3] and T. Dillon[4]

Abstract In this paper, we address the issues of using OWL to model the knowledge captured in relational databases. Some types of knowledge in databases cannot be modeled directly using OWL constructs. Two alternative approaches are proposed with examples of two types of knowledge. Firstly the data value range constraint and secondly the calculation knowledge representation. The first approach to the problem is to conceptualize the data range as a new class and the second solution to the problem is proposed, based on utilizing software agent technology. Examples with OWL code and implementation code are given to demonstrate the problems and solutions.

1 Introduction

With the increasing trend of collaborations amongst organizations and business needs for sharing and publishing their products information, information and knowledge held in vast number of databases are demanded to be shared and integrated without organizational and application boundaries. However, databases are enterprise and application dependant in that their design and development are subjected to a particular business problem domain of an organization. This has prevented the databases from being shared and integrated in an open environment.

Ontology-based technologies provide a feasible approach to this problem. Ontology-based technologies promote knowledge sharing and integration by formally and explicitly defining the meanings and associations of information and data. An ontology is defined as "*a formal, explicit specification of shared conceptualization*" [1-3]. Ontologies allow specially designed software agents to automatically process and integrate information from distributed sources. Many approaches have been proposed to transform the knowledge embedded in databases, particularly in relational databases, into ontologies [4-9]. The transformation process involves database reverse engineer-

1 2 3 4.Shuxin Zhao, Dr. Pornpit Wongthongtham, Prof. Elizabeth Chang, Prof. Tharam Dillon

Digital Ecosystems & Business Intelligence Institute, Curtin University of Technology, Australia.

email: {s.zhao, p.wonthongtham, e.chang, tharam.dillon}@curtin.edu.au

ing for acquiring the implicit knowledge from databases and involves mapping the extracted knowledge onto an ontology language. OWL [10], particularly OWL DL, as the WWW consortium recommendation for Semantic Web, has gained the popularity as the target ontology language. Hereafter, we refer OWL in this paper to OWL DL, as it's the most practical one among the three sub-languages for Semantic Web.

However, there is a critical issue of using OWL, to fully and accurately represent the knowledge captured in relational databases. Although there are many similarities between an ontology and a conceptual data model of a database, such as UML or EER model, there are many practical issues when mapping the knowledge captured in a conceptual model onto an OWL ontology. For example, there are three common types of relationships between concepts we model in an UML model, namely, generalization/specialization, aggregation and composition and association. While generalization/specialization can be modeled straightforwardly using OWL hierarchical mechanism i.e. *Class* and *Subclass*, *Property* and *Subproperty*, the aggregation/composition relationship cannot be represented directly using OWL elements. There are also other types of knowledge captured by relational databases that we found hard to represent using OWL constructs such as the value range restrictions on an attribute, and the functional dependency among several attributes of one or more tables which captures some sort of relationships between attributes rather than concepts.

In this paper, we present two alternative solutions to tackle this OWL modeling issue, namely, conceptualization approach and software agent based approach. Two specific examples are used to demonstrate each of the approaches respectively: firstly the problems of modeling the data value range constraint; secondly, the problem of modeling mathematic calculation knowledge, whose operands are derived from attributes of one or more concepts, which represents relationships between these attributes. Our motivation is to reveal some ideas of extending the expressiveness of OWL in the mean time to retain computational completeness of the ontology model, thus to make OWL more adaptable to various domain knowledge representations.

The rest of this paper is organized as follows: Section 2 reviews related work on these issues; Section 3 describes the problems in details with examples; followed by Section 4 demonstrating the solutions to the problems with code example; last in Section 5, we conclude the paper and indicate future work.

2 Related Work

OWL provides powerful mechanisms to enhance reasoning about the *Classes* and relationships amongst *Classes* but not for representing and reasoning relationships between *Properties*. The OWL modeling issue originates from its design pursuit of the trade-off between expressiveness and scalability of a language. Most important kinds of knowledge are supported in OWL, particularly for sub-assumption and classification, while the computational completeness and decidability must also be retained [11, 12]. As a consequence, the OWL is designed to be maximum expressive without being undecidable. This has resulted in the expressiveness limitations amongst other OWL weakness which are identified in [13]. One of W3C's solutions to this problem is to introduce *Rules* (*RIF*) [14] which aims to provide greater expressiveness in con-

junction with RDF/OWL, typically, to provide a richer language for representing dependencies between *Properties* rather than *Classes*. *RIF Core Design* working draft has been released in Oct 2007. One plausible drawback to this Rule-based solution is that the knowledge needs to be encoded by more than one or two languages in order to represent the full domain.

Besides of the above, there is not much work that has been reported on addressing the issues of the knowledge representation with OWL. Stojanovic et al. [5] mentioned that some database related dynamic knowledge embedded in SQL stored procedures, triggers and built-in functions cannot be mapped to RDF.

3 Problem Description with Examples

In this section, we describe the two specific types of knowledge that cannot be modeled directly using OWL constructs in order to demonstrate the idea of tackling the above mentioned modeling issues. Solutions to the problems are given in the following section.

3.1 Data Value Range Modeling Problem in OWL

The first type of knowledge that we mentioned in the introduction section that cannot be modeled directly using constructs, which are specified in OWL DL, is the constraint on data value range. Data value range constraint is very common to various domains. For example, a company recruitment statement contains a minimum age and a maximum age requirement and a bank product requests a minimum and a maximum amount of deposit over a period such as monthly. This refers to data value range in database development. This kind of data constraint can be obtained from database schema, application source code through validation and SQL queries. It, however, cannot be directly represented using any constructs specified in OWL DL. One example of the recruitment requirement for the employee's age constraint in a company, named ABC, can be expressed as the formula:

```
ABCEmployee (18 < age < 65)
```

In OWL DL, if we define a Class namely *Employee*, with a *DatatypeProperty* namely *age* shown as in the OWL definition below:

```
<owl:Class rdf:ID="Employee"/>
<owl:DataTypeProperty rdfID="age">
       <rdfs:domain rdfresource="#Employee" />
       <rdfs:range rdfresource="&xsd;integer" />
<owl:DatatypeProperty>
```

We may further add constraints such as the cardinality on the *age* property, but no any other elements defined in OWL for property restrictions, such as *allValueFrom* and the set operator like *unionOf* and *intersectionOf*, can be used to model the simple

value range constraint. We therefore need other means to represent this kind of knowledge in OWL ontologies.

3.2 Calculation Knowledge Representation Problem in OWL

The second type of knowledge cannot be modeled directly using OWL constructs is the general calculation knowledge. An arithmetic calculation consists of operands and arithmetic operators such as addition, subtraction, multiplication and division. Operands in a calculation are often derived from columns of tables in a database or from properties of Classes in an ontology. The result of a calculation, in the mean time, is assigned to a column or a property. This represents associations amongst properties rather than classes. It may also represent the dynamic knowledge which is generated at run time in a given application. This type of knowledge is usually defined in SQL queries such as stored procedures or application source code when validating new data entry to ensure data consistency. One example of this type of knowledge is the calculation of total cost including GST tax of a purchase. The cost is calculated based on three properties: the *"quantity"* of the product in the purchase, the *"price"* of the product excluding GST tax and current *"GST tax rate"*. It can be expressed as the following formulas:

```
SubTotal = itemQuantity * singleUnitPrice
Tax = SubTotal * GSTRate
TotalCost = SubTotal + Tax
```

In OWL, there is no constructs defined for modeling this type of associations among properties from one or more Classes.

4 Approach

For the modeling problems stated in the previous section, we propose two alternative approaches to tackle the issues. They are described and demonstrated with sample code in this section.

4.1 Conceptualization of Data Value Range Constraint in OWL

As OWL does not provide any constructs for restricting value range on *DatatypePropertie*, we cannot represent this constraint directly in the way that we specify it in a programming language or in a database management system. However, we can model the value range constraint by conceptualizing it into a new Class. The conceptualization actually explicitly reflects the semantics of the data restriction because the general concept *Age* of human being is different from the concept *minimum age* and *maximum age* in a company recruitment requirement. We demonstrate the solution to the first problem defined in section 3.1 as in List 1.

In the OWL ontology List 1, the constraint on employee's age is conceptualized as a new Class *"EmploymentAge"*. It has two *DatatypeProperties*: *"minAge"* and *"maxAge"*. There is one individual created for ABC company recruitment requirement called *"ABCEmploymentAge"* whose *"minAge"* is 18 and *"maxAge"* is 65. The property *"age"* of the Class *"Employee"* can therefore be defined as an *ObjectProperty* whose range is of the class *"EmployeeAge"*. If there are individuals of ABC company employee, their age must be between 18 and 65. One key point to this solution is that this conceptualization must be transformed or mapped in implementation. Likewise, other types of knowledge can also be conceptualized in this way.

```
<owl:Class rdf:ID="EmploymentAge"/>

 <owl:DatatypeProperty rdf:ID="maxAge">
    <rdfs:domain rdf:resource="#EmploymentAge"/>
    <rdfs:range rdf:resource="&xsd#int"/>
  </owl:DatatypeProperty>

 <owl:DatatypeProperty  rdf:ID="minAge">
    <rdfs:range rdf:resource="&xsd#XMLSchema#int"/>
    <rdfs:domain rdf:resource="#EmploymentAge"/>
  </owl:DatatypeProperty>

  <EmploymentAge rdf:ID="ABCEmploymentAge">
     <maxAge rdf:datatype="&xsd#int">18</maxAge>
     <minAge rdf:datatype="&xsd#int">65</minAge>
  </EmploymentAge>

 <owl:Class rdf:ID="Employee"/>

 <owl:ObjectProperty rdf:ID="employmentAge">
    <rdfs:domain rdf:resource="#Employee"/>
    <rdfs:range rdf:resource="#EmploymentAge"/>
 </owl:ObjectProperty>
```

List 1 conceptualization of data value range constraint in OWL

4.2 Software Agent-Based Knowledge Representation Approach

The second approach is to utilize software agent technology. Software agent technology has been in extensive discussion for many years but it is perhaps recently that it has been attracting much attention of exploitation in the emergence of the Semantic Web. Basically software agents are components in an application that are characterized by among other things autonomy, pro-activity and an ability to communicate [15]. Autonomy means that agents can independently carry out complex and long term tasks. Pro-activity means that agents can take initiative to perform a given task without human intervention. Ability to communicate means agents can interact with other agents or other components to assist to achieve their goals.

In this paper we implement software agents using JADE (Java Agent Development framework), an agent-oriented middleware [16, 17]. The reason we use JADE is simply because it facilitates development of complete agent-based applications and it is written in well known object-oriented language, Java. More details of JADE can be found on its website (http://jade.tilab.com).

Basically in this paper, we utilize JADE agent technology to help define the knowledge of calculation. A JADE agent is identified under FIPA specifications [18] by an agent identifier. A task can be defined for an agent to carry out. Agent action defines the operations to be performed. Agent communication according to FIPA specifications [18] is the most fundamental feature of software agents. Format of messages is compliant with that defined by FIPA-ACL message structure.

For the calculation knowledge described in section 3.2, we can define it in the following formula.

```
Total Cost = Price * Quantity * (1 + GST Rate / 100)
```

Ontologies are typically specific to a given domain. For the above formula we specify to a product trading domain which would not be the same as in a payroll system. Thus product concept could have properties of name, barcode, etc. Agents then have some shared understanding with the product concept and its properties. There may be two products named the same. In order to unequivocally identify a product, it may be necessary to specify barcode.

According to the FIPA specifications [18], when agents communicate, product information representation is embedded inside ACL messages. Because JADE agents are Java-based, the information can be represented using objects.

In order to exploit agent and ontology technology to support and allow agents to discourse and reason about facts and knowledge related to a given domain, we specify the approach into 3 steps.

- Define concepts in an ontology. In the purchase example, it includes *Product* and *Purchase* concepts.
- Develop proper Java classes for the above two concepts in the ontology.
- Define the calculation formula by hard-coding it.

In order to illustrate defined concepts of *Product* and *Purchase* in an ontology, we model *Product* and *Purchase* knowledge representation shown in Figure 1. Figure 1 (A) shows *Product* concept and Figure 1 (B) shows *Purchase* concept. Ontology class *Product* has datatype properties of *name* and *barcode* both related to a string type. Ontology class *Purchase* has object properties of *item* related to the ontology class *Product*. The ontology class *Purchase* also has datatype properties of *price* related to a float type and *quantity* and *tax_rate* related to an integer type.

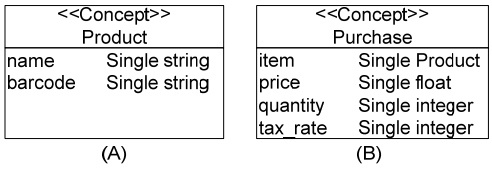

Figure 1 Product and Purchase concepts in ontology modelling

We reuse schema classes available in JADE *PredicateSchema*, *AgentActionSchema*, and *ConceptSchema* included in the *jade.content.schema* package to define the structure of each type of predicate, agent action, and concept respectively [17]. In the example, we can model the domain including one concept (Product), one predi-

cate (Purchase – to apply to a product) and one agent action (Calculate – to calculate total cost including tax).

Since the ontology is shared among agents, *TradeOntology* class is placed in an ad-hoc package, ontology. The ontology defined in Java is shown in List 2.

```
   package TradingPackage;

   import jade.content.onto.*;
   import jade.content.schema.*;
   import jade.util.leap.HashMap;
   import jade.content.lang.Codec;
   import jade.core.CaseInsensitiveString;

   public class TradeOntology extends jade.content.onto.Ontology  {
     //NAME
     public static final String ONTOLOGY_NAME = "Trade";
     // The singleton instance of this ontology
     private static ReflectiveIntrospector introspect = new Reflec-
tiveIntrospector();
     private static Ontology theInstance = new TradeOntology();
     public static Ontology getInstance() {
        return theInstance;
     }

      // VOCABULARY
       public static final String PURCHASE_ITEM="Item";
       public static final String PURCHASE_QUANTITY="Quantity";
       public static final String PURCHASE_TAX_RATE="Tax_Rate";
       public static final String PURCHASE_PRICE="Price";
       public static final String PURCHASE="Purchase";
       public static final String CALCULATOR="Calculator";
       public static final String CALCULATE="Calculate";
       public static final String PRODUCT_NAME="Name";
       public static final String PRODUCT_BARCODE="Barcode";
       public static final String PRODUCT="Product";

     /*  Constructor   */
     private TradeOntology(){
       super(ONTOLOGY_NAME, BasicOntology.getInstance());
       try {

       // adding Concept(s)
       ConceptSchema productSchema = new ConceptSchema(PRODUCT);
       add(productSchema, TradingPackage.Product.class);

       // adding AgentAction(s)
       AgentActionSchema calculateSchema = new AgentAction-
Schema(CALCULATE);
       add(calculateSchema, TradingPackage.Calculate.class);

       // adding AID(s)
       ConceptSchema calculatorSchema = new ConceptSchema(CALCULATOR);
       add(calculatorSchema, TradingPackage.Calculator.class);

       // adding Predicate(s)
       PredicateSchema purchaseSchema = new PredicateSchema(PURCHASE);
       add(purchaseSchema, TradingPackage.Purchase.class);

       // adding properties
       productSchema.add(PRODUCT_BARCODE, (Term-
Schema)getSchema(BasicOntology.STRING), ObjectSchema. MANDATORY);
       productSchema.add(PRODUCT_NAME, (Term-
Schema)getSchema(BasicOntology.STRING), ObjectSchema.OPTIONAL);
       purchaseSchema.add(PURCHASE_PRICE, (Term-
Schema)getSchema(BasicOntology.FLOAT), ObjectSchema.MANDATORY);
```

```
      purchaseSchema.add(PURCHASE_TAX_RATE, (Term-
Schema)getSchema(BasicOntology.INTEGER), ObjectSchema.MANDATORY);
      purchaseSchema.add(PURCHASE_QUANTITY, (Term-
Schema)getSchema(BasicOntology.INTEGER), ObjectSchema.MANDATORY);
      purchaseSchema.add(PURCHASE_ITEM, productSchema, ObjectSchema.
MANDATORY);

     }catch (java.lang.Exception e) {e.printStackTrace();}
    }  }
```

List 2 Trade Ontology defined in Java

```
package TradingPackage;

import jade.content.*;
import jade.util.leap.*;
import jade.core.*;

public class Product
     implements Concept {

// Barcode
 private String barcode;
 public void setBarcode(String
value){
      this.barcode=value; }

public String getBarcode() {
    return this.barcode;
}

// Name
 private String name;
 public void setName(String value) {
      this.name=value;
 }
 public String getName() {
      return this.name;
 }
}
```

List 3 Product concept defined in Java

```
package TradingPackage;

import jade.content.*;
import jade.util.leap.*;
import jade.core.*;

  public class Purchase imple-
ments Predicate {

  // Price
   private float price;
   public void setPrice(float
value) {
     this.price=value;
   }
   public float getPrice() {
     return this.price;
   }

  // Tax_Rate
   private int tax_Rate;
   public void setTax_Rate(int
value) {
     this.tax_Rate=value;
    }

   public int getTax_Rate() {
     return this.tax_Rate;
   }

  // Quantity
   private int quantity;
   public void setQuantity(int
value) {
     this.quantity=value;
   }
   public int getQuantity() {
     return this.quantity;
   }

  // Item
   private Product item;
   public void setItem(Product
value) {
     this.item=value;
   }
   public Product getItem() {
     return this.item;
   }
}
```

List 4 Purchase concept defined in Java

The schemas for *product*, *purchase*, *calculate*, and *calculator* concepts are associated with *product.java*, *purchase.java*, *calculate.java*, and *calculator.java* classes respectively. Each property in a schema has a name and a type. For example, in the *product* schema, *barcode* has its type as string. Every product must have barcode as

declared as MANDATORY. Similarly, value for properties *item*, *price*, *quantity*, and *tax rate* cannot be null because when the purchase is made these values are mandatory. Validation is made by throwing an exception if the value of mandatory properties is null.

The *product* concept could be defined specifically to particular products e.g. books, CDs for more specific trading. Properties of the product concept i.e. *name* and *barcode* will be inherited to books and CDs. Book and CDs concepts can have their own specific properties e.g. the CDs concept might have *tracks* property and books might have *authors* property and so on.

Java classes, associated with the product concept and the purchase predicate in the example, are shown in List 3 and List 4 respectively.

Agent action associates with the agent identifier which is intended to perform action for this example to calculate total cost included tax. Calculation can be hard coded getting value from object of class purchase i.e. price, quality, and tax rate.

For example a product of $200 price, 2 quantity, and 10% tax rate would have expression as following:

```
((action (agent-identifier :name calculator) calculate (product :name "xxx" :barcode "01211") purchase (product :name "xxx" :barcode "01211") 360)
```

Alternatively, we can also specify in class purchase as the attribute of *TotalCost* shown as in List 5 below.

```
// Total Cost
   private float TotalCost;
   public float getTotalCost() {
     return this.price * this.quality * (1 + tax_Rate / 100);
}
```

List 5 The formula defined in Java

One advantage using software agent based approach, in comparison to the conceptualization approach, is that it has already been realized in software agent definition which does not require further implementation code.

5 Conclusion

In this paper we addressed the practical problems associated with knowledge representation in OWL. OWL specifications provide many mechanisms for defining restrictions and associations among *Classes* but not for *Properties*. We have presented two types of knowledge, which are common to various domains, but cannot be modeled directly using constructs specified in OWL. To tackle this knowledge presentation gap in OWL, we have proposed two alternative solutions to the problems. One is to conceptualize the knowledge such as the data value range constraints and the other is to use other existing technology such as software agents to encode and convey the

knowledge. As we gave calculation knowledge example defined in the formula for experimentation. With the knowledge defined in the ontology the software agents are able to use calculation knowledge to define a new knowledge (i.e. the total cost). Its prototype is still under development and need to extend in different fields. We do not intend to list all OWL modeling problems rather we aim to provide some useful hints to other likewise knowledge representation issues with OWL that have yet to be resolved.

References

1. R. Studer, V. R. Benjamins, and D. Fensel, "Knowledge engineering: Principles and methods," Data & Knowledge Engineering, vol. 25, pp. 161-197, 1998.
2. W. Borst, "Construction of Engineering Ontologies," University of Twente, Enschede, 1997.
3. T. R. Gruber, "A Translation Approach to Portable Ontology Specifications," Knowledge Acquisition, vol. 5, pp. 199-220, 1993.
4. V. Kashyap, "Design and creation of ontologies for environmental information retrieval," presented at the 12th Workshop on Knowledge Acquisition, Modeling and Management, Alberta, Canada, 1999.
5. L. Stojanovic, N. Stojanovic, and R. Volz, "Migrating data-intensive Web Sites into the Semantic Web," presented at the 17th ACM symposium on applied computing (SAC),, SAC, 2002.
6. R. Meersman, "Ontologies and Databases: More than a Fleeting Resemblance," presented at OES/SEO Workshop Rome, Rome, 2001.
7. I. Astrova, "Reverse engineering of relational database to ontologies," presented at First european Semantic Web symposium, ESWS, Heraklion, Crete, Greece, 2004.
8. M. LI, X.-Y. DU, and S. WANG, "Learning ontology from relational database," presented at the Fourth International Conference on Machine Learning and Cybernetics, Guangzhou, China, 2005.
9. S. Zhao and E. Chang, "Mediating Databases and the Semantic Web: A methodology for building domain ontologies from databases and existing ontologies," presented at SWWS'07- The 2007 International Conference on Semantic Web and Web Services, Las Vegas, Nevada, USA, 2007.
10. W3C, "Ontology Web Language," vol. 2006, M. K. Smith, C. Welty, and D. L. McGuinness, Eds.: WC3, 2004.
11. W3C, "OWL Web Ontology Language Use Cases and Requirements," vol. 2007, J. Heflin, Ed., 2004.
12. N. Shadbolt, W. Hall, and T. Berners-Lee, "The Semantic Web Revisited," IEEE Intelligent Systems, pp. 96-101, 2006.
13. D. Reynolds, C. Thompson, J. Mukerji, and D. Coleman, "An assessment of RDF/OWL modelling," Digital Media Systems Laboratory, HP Laboratories Bristol 28 Oct 2005.
14. W3C-RIF, "Rule Interchange Format Working Group," vol. 2007: W3C, 2007.
15. M. Wooldridge, Introduction to MultiAgent Systems. , 1st ed: John Wiley & Sons., 2002.
16. F. Bellifemine, "JADE Java Agent DEvelopment Framework," Telecom Italia Lab: Torino, Italy., 2001.
17. F. Bellifemine, G. Caire, and D. Greenwood, Developing Multi-Agent Systems with JADE: John Wiley & Sons Ltd., 2007.
18. F. Bellifemine, A. Poggi, and G. Rimassa., "JADE: a FIPA2000 compliant agent development environment," presented at The fifth International Conference on Autonomous Agents, Montreal, Quebec, Canada, 2001.