Exporting Knowledge Bases into OWL

Vinay K. Chaudhri, Bill Jarrold, John Pacheco

Artificial Intelligence Center, SRI International, Menlo Park, CA 94025
{vinay.chaudhri, william.jarrold, john.pacheco}@sri.com

Abstract. We present our experience in exporting a knowledge base (KB). Specifically, we discuss the translations of the representation of scalar, cardinal, and categorical values, tasks, built-in data types, and collections. These examples could be illustrative for others trying to use OWL in their work. We also raise design questions that could provide fodder for discussion at the workshop.

1 Introduction

Many knowledge representation and reasoning systems do not use OWL as their native format, but export to OWL to achieve interoperability. In this paper, we consider one such case study. Our experience raises some interesting design questions. For example, do the current sublanguages in OWL provide a direct sweet spot for practical applications?

We are conducting this work in the context of CALO (Cognitive Assistant that Learns and Organizes), a multidisciplinary project funded by DARPA to create cognitive software systems that can reason, learn from experience, be told what to do, explain what they are doing, reflect on their experience, and respond robustly to surprises (See http://caloproject.sri.com/ for more information.)

2 Problem Setup

We are using the Knowledge Machine (KM) system to develop the KB [1]. Our KB uses the component library or CLib [2], a generic domain-independent KB, as its upper ontology. We extended CLib by adding representations for the office domain, for example, People, Meetings, Tasks, Organizations, and Projects [3].

Many components of CALO needed to incorporate the ontology into their code base. Therefore, we needed to export the ontology from KM into some Interlingua that could be loaded by other system modules. A key system module that needed to use the ontology was IRIS. IRIS is a semantic desktop application [4] that is built on Jena graphs and uses

OWL as its native language. We chose OWL as an exchange language because it offered most of the features needed for an interchange language for the project.

We implemented the translator from KM to OWL as a Lisp program. The translator takes a KB expressed in KM and produces the OWL equivalent to the extent possible. KM expressions, for example, rules, are not easily expressed in OWL and are omitted from the translation. We load the output of the translator into Protégé [5], which allows us to browse the result and has the added benefit of ensuring that we are generating legal OWL. If we encounter errors while loading the translation output into Protégé, we use the Vowlidator[1] or the WonderWeb[2] OWL Ontology Validator. We process the ontology using OWLDoc,[3] so that the ontology documentation can be viewed by end users.

Since the representation language in KM is more expressive than OWL, and OWL has three sublanguages—OWL-Lite, OWL-DL, and OWL-full— we had to make choices on which sublanguage of OWL to use in the translation. After we present the translations of relevant features of KM into OWL, we step back and analyze the features called for by the application. Based on the observation that we needed to use some subset of features from OWL-DL and OWL-full we speculate about whether we should consider a sublanguage of OWL that cuts across the three sublanguages.

3 Description of Translation

We now discuss aspects of the OWL translation: (1) representing scalar, cardinal, and categorical values; (2) representing task knowledge; (3) experience using built-in data types; and (4) representing collections.

3.1 Representing Scalar, Cardinal, and Categorical Values

CLib distinguishes two kinds of values: *categorical* and *scalar/cardinal* [2]. The fillers of properties such as `color`, `duration`, or `length` are an instance of class `Property-Value`. The range of each such slot has some subclass of `Property-Value` as its range (e.g., `Color-Value`, `Duration-Value`, `Length-Value`). Several different property slots may share the same range. For example, length, width, and distance all have `Length-Value` as their range.

Representing Scalar Values. A scalar value ascribes a symbolic value in relation to a reference class, for example, hot for a drink, or small for a house. The values are ordered; for example, hot involves a higher temperature than warm. Although an object may have

1 See http://projects.semwebcentral.org/projects/vowlidator/

2 See http://wonderweb.semanticweb.org/

3 See http://www.co-ode.org/downloads/owldoc/co-ode-index.php.

only one value for a given property (e.g., height is single valued), that value may be associated with multiple scalar values when there are multiple reference classes, e.g., a person has only one value for height. However, that value may be considered “tall” with reference to the class Person but “short” with reference to the class representing basketball players. CLib reifies the scale value, and allows one to assert the magnitude of the reified value, e.g., we can represent *Ana is short for a Person* as follows:

```
<Person rdf:ID="Ana">
 <height>
  <Length-Value>
   <scalar-value>
    <Scalar>
     <scalar-constant rdf:resource="#Short"/>
     <reference-class rdf:resource="#Person"/>
    </Scalar>
   </scalar-value>
  </Length-Value>
  </height>
</Person>
```

We have reified the value of `height` to `Anas-Height`. The value of the `height` slot has dimension `Length,` and therefore, we define `Anas-Height` as an instance of `Length-Value.` The relation `scalar-constant` represents the magnitude of a `Length-Value` with respect to a reference class such as `Person`.
In KIF and KM, it is possible to define `scalar-value` as a ternary relation instead:

```
(scalar-value-with-reference-to Height-1 Short Person)
```

Since OWL does not allow ternary relations, we reify the arguments of what would have been a ternary relation, and specify them using `scalar-constant` and `reference-class` [6].

Representing Cardinal Values. A cardinal value is a quantity in some dimension. A unit of measure is associated with each dimension, for example, *days*, *grams*, or *degrees Fahrenheit*. To represent unitless quantities such as slope, CLib provides a unit called `UoM-Unitless`. We can represent that Ana has a height of 1.2 m as follows:

```
<Person rdf:ID="Ana">
 <height>
  <Length-Value>
   <cardinal-value>
    <Cardinal>
      <xsd:float>
      <rdf:numeric-value>1.2</rdf:numeric-value>
      </xsd:float>
     <unit rdf:resource="#Meter"/>
    </Cardinal>
   </cardinal-value>
  </Length-Value>
 </height>
</Person>
```

The height values are instances of `Length-Value` as they are for `distance`, `width`, and so on. We reify an instance of `Cardinal` (rather than a `Scalar`) and use `numeric-value` (rather than `scalar-constant`) to specify its value.

Representing Categorical Values. Categorical values are symbolic in that they do not sensibly appear on any scale or continuum. There is no ordering to the categorical values. For example, *color* is a categorical value. There is no natural ordering of its symbolic values: *blue*, *orange*, and so on. Although it is possible to assign arbitrary numeric values to the symbolic constants (such as the RGB encoding of colors), it is not a feature of the colors that they are ordered. Outside of some arbitrary imposed ordering, it does not make sense to say that “blue is greater than orange”, or “something green has a higher/greater color than something red”. Suppose we wish to state *Ana’s car is black.* We can state

```
<Person rdf:ID="Ana">
 <possesses>
  <Car rdf:about=”Car-35”>
    <colorIs>
     <Color-Value>
     <categorical-value>
      <categorical-constant rdf:resource=”#Black”/>
     </categorical-value>
    </Color-Value>
   </colorIs>
  </Car>
 </possesses>
</Person>
```

3.2 Representing Task Knowledge

The CALO system makes an extensive use of tasks. For example, the user may instruct the system to purchase a piece of equipment, arrange a meeting, or remind the user of an important deadline. The CALO system has an extensive collection of process models that can execute the tasks on behalf of the users. Some of these process models are engineered by hand, while others are learned by the system either by observation or by instruction. We represent the executable tasks within a procedural reasoning and execution system called SPARK [7]. SPARK provides an expressive language for encoding the execution of procedures. A discussion on the relationship between the procedure language of SPARK and other process languages such as OWL-S is available elsewhere [8]. With each procedure, we can associate a declarative task that specifies the parameters of the procedure, and organizes the tasks into a taxonomy. The KM system is well suited for the representation of such declarative knowledge about tasks, but we faced several challenges in translating that representation into OWL.

The parameters to an executable task often have meta-properties associated with them. For example, some of the parameters are input to the procedure, while others are output. Similarly, some are required parameters, while others are optional. Given a task, there is a frequent need for queries such as what are the input or output parameters. We consider two approaches for meeting this requirement, *property based* and *instance based*. Both the approaches that we discuss here are allowed in KM and CLIB, but they map to different sublanguages of OWL, and present tradeoffs in terms of which sublanguage to use.

The first approach, *property based*, is to define slots such as `inputProperty` and `requiredInputProperty` and assert that `requiredInputProperty` is a subproperty of `inputProperty`. Consider as an example the class `ExtractMeetingTask`—the task of extracting proposed meeting information such as location, participants, and start/end times from an email message. Within the above framework, `hasEmail` would be an input parameter and would be required.

```
<owl:ObjectProperty rdf:ID=”optionalOutputPropertyIs”>
 <rdfs:subPropertyOf>
  <owl:ObjectProperty rdf:about=”outputPropertyIs”/>
  </rdfs:subPropertyOf>
 <rdf:type rdf:resource=”&owl;AnnotationProperty”/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID=”requiredInputPropertyIs”>
 <rdfs:subPropertyOf>
  <owl:ObjectProperty rdf:about=”inputPropertyIs”/>
</rdfs:subPropertyOf>
    <rdf:type rdf:resource=”&owl;AnnotationProperty”/>
  </owl:ObjectProperty>

<owl:Class rdf:ID="ExtractMeetingTask">
 <optionalOutputPropertyIs>
  <owl:FunctionalProperty rdf:ID="meetingParticipantsAre"/>
 </optionalOutputPropertyIs>
</owl:Class>
```

In the instance-based approach, we define classes such as `InputParameter` and `OutputParameter.` For each argument to a procedure, we would then define an instance of the relevant class, and assert properties on that instance to add detail [6].

Consider, for example, the instance-based analog of the `hasEmail` parameter. Instead of using the slot `hasEmail`, we would reify an instance, call it Parameter007, about which we will make the following assertions in OWL full:

```
<InputParameter rdf:ID=”Parameter007”>
 <rdf:type>
  <owl:Class rdf:about=”#OptionalParameter”>
 </rdf:type>
</InputParameter>
```

One advantage if instance-based approach is that one can express the fact that no input parameter is also an output parameter simply by asserting

```
<owl:Class rdf:ID="InputParameter">
 <owl:disjointWith>
  <owl:Class rdf:about=”#OutputParameter”>
 <owl:disjointWith>
</owl:Class>
```

There is no way to express this fact using the property-based approach within OWL-DL. The second advantage is that one can easily represent and reason about meta-properties about the parameter, e.g., one can express the position of a given parameter within the task signature. Furthermore, one can express the constraint that a parameter can occupy at most one position.

```
<InputParameter rdf:about=”Parameter007”>
 <parameterPositionIs>1</parameterPositionIs>
</InputParameter>
<owl:DatatypeProperty rdf:ID="parameterPositionIs">
 <rdf:type rdf:resource="&owl;FunctionalProperty"/>
</owl:DatatypeProperty>
```

The advantage of the property-based approach is that it leads to more compact representation and involves reifying fewer new objects. In the current ontology, we support both these representations. Ideally, we would have liked to be able to represent the translation between the two representations in OWL, but that requires using rules that are outside the scope of OWL.

3.3 Using Built-in Types

We mapped built-in data types in KM to standard types in OWL, e.g., if a slot had a cardinality constraint defined as N to 1 in CLib, we defined it to be an instance of Functional Property in OWL. If it had a cardinality constraint of 1 to 1, we defined it to be an instance of OWL class *FuncationalProperty* and *InverseFunctionalProperty.*

OWL distinguishes two kinds of slots: *DatatypeProperty* and *ObjectProperty*. KM does not make this distinction. It was straightforward to compute this distinction by checking if the range of a slot is an XSD data type, for example, *String* or *Number*.

KM uses the type Number as a generic type for all numbers. There is no straightforward mapping from the type Number into OWL. There is no class that is a least common ancestor of all the classes that represent numbers. We believe this to be a limitation of the current design of the data types in OWL.

In OWL, one cannot define subclasses of primitive data types such as STRING. There were several compelling situations where such a feature was critical. For example, a learning algorithm may deduce that a string such as "94025-3493" is an instance of the class of strings representing U.S. Postal Codes, and that a string such as "650-555-1212" is an instance of the class representing phone numbers in the United States. Once deduced, there is a need to enforce it as a constraint on the legal values of postal codes and phone numbers. To support this requirement, we introduced a collection of classes termed *Pseudo Ranges* that were not a subclass of the built-in OWL class String. For example, *PostalCodeString* is a pseudo range class that is a string, and defines legal strings for U.S. Postal Codes. This was not an isolated example; our KB has numerous such examples.

The pseudo range approach has some disadvantages. First, even though one may make an assertion that the pseudo range of the a`ddressPostalCodeIs` is `PR-ZipCode`, such an assertion does not invoke any type checking. Second, this approach allows one to reify an instance for every ZIP code. This can get computationally expensive. There are some reasoners, such as Pellet,[4] that allow extending the built-in types such as STRING, but their extensions are not interoperable and not a part of the OWL standard.

3.4 Representing Collections

Collection data types are sets, bags, lists, and tuples. The CALO application has a frequent need to use different collection types. For example, the attendees of a meeting need to be represented as a set. The result of searching a collection of documents is a ranked order list. If a user queries for the prices of homes in an area, the answer is a collection of tuples of length two. While computing the median price of homes in a region, we need to construct a bag of prices in which duplicates are retained.

The container constructs from the RDF vocabulary—rdf:List and rdf:nil—are unavailable in OWL-DL because they are used in the RDF serialization of OWL in [9]. Although rdf:Seq is not illegal, and one could get around the unavailability of rdf:List and `rdf:nil` within OWL by defining equivalent constructs in the OWL name space, the container representation in RDF has the following disadvantages: (1) The elements in a container are defined using the relations `rdf:_1, rdf:_2,` and so on that have no formal definition in RDF. Using them for the purpose of reasoning will require us to

[4] See: http://www.mindswap.org/2003/pellet/index.shtml

define and enforce the properties of these relations. (2) It is not possible to define a container that has elements only of a specific type. (3) For updating a specific element in a container in a remote source, one is forced to transmit the whole container. (4) It is not possible to associate provenance information with the elements in a container.

Our approach for modeling collection types introduces additional vocabulary in OWL. We introduce a class `LinkedList` to represent instances of a linked list. The `first-element` is a functional property that denotes the first element in a list. The relation `restOfListIs` is a property that captures the recursive structure of the list. We use a distinguished property value `LinkedListNull` to denote the list termination.

```
<LinkedList>
   <first-element rdf:resource="Sally" />
   <restOfListIs>
    <LinkedList>
    <first-element rdf:resource="Leigh"/>
    <restOfListIs rdf:resource="LinkedListNull" />
   </LinkedList>
   </restOfListIs>
  </LinkedList>
```

We introduce a class called `Bag` to represent the instances of a bag collection type. We use the property `element` to represent the elements of a bag, and the instance of class `BagElement` to hold the elements of a bag. The relation `object` on `BagElement` is a functional relationship. We need to introduce the element `BagElement` so that we can represent duplicate values. Without the use of `BagElement,` multiple identical values of <element> will be eliminated because by default they are treated as a set.

```
<Bag rdf:ID=”Bag01”>
  <element>
   <BagElement rdf:ID=”BagElement01”>
    <object rdf:resource=”Sally”>
   </BagElement rdf:ID>
  </element>
  <element>
   <BagElement rdf:ID=”BagElement02”>
    <object rdf:resource=”Leigh”>
   </BagElement rdf:ID>
  </element>
  <element>
   <BagElement rdf:ID=”BagElement03”>
    <object rdf:resource=”Sally”>
   </BagElement rdf:ID>
  </element>
 </Bag>
```

To represent tuples, we introduce the class. The `element` property on `Tuple` has as its value the instances of the class `TupleElement`. The instances of the class `TupleElement` hold the individual tuples. The property `object` on `TupleElement` is functional, and we introduce an additional property `positionInTupleIs` to capture the position of the tuple element in the overall tuple. In principle, we can implement a tuple as a subclass of linked list by adding a restriction on the length property. But, that is ontologically not correct. In a linked list, the second element of a list is a linked list or `nil`. In a tuple there is no requirement for the second element to be a linked list or nil.

5 Discussion Topics for the Workshop

Design of built-in data types: We identified two limitations of the built-in data types of OWL. First, there is no generic Number class in OWL, which made translation from KM to OWL difficult. Omission of such a general type appears to be an oversight to us because a generic Number type is essential for interoperability. Second, the inability to define subclasses of built-in classes is a serious limitation. We illustrated a large number of subclasses of String that we needed to define. Should we rethink the design of built-in types in OWL? Is there a better way to represent the same knowledge?

Suggestions for the OWL Best Practices Working Group: The Working Group provides will require several worked-out representations in OWL to support the OWL user community. There is already some interest from the members of this task force for best practices for representing units and measures. The translation presented in this paper can be a useful input to the Working Group. While there is no explicit expression of interest in representations for collections and tasks, we believe that they are of sufficiently broad interest that the Working Group should consider providing best practices for them.

Sublanguages of OWL: We can analyze the three sublanguages by asking the following questions: Could we have used OWL-Lite? Could we have used OWL-DL? Which features of OWL-DL were most used? Which features of OWL-full were used? What is an appropriate language for the CALO application?

The units and measures representation that we consider in this paper can be expressed in OWL-Lite. The instance-based approach for representing task parameters can be represented in OWL-Lite with the exception of the disjoint-ness assertion between the classes, which requires OWL-DL. The problems we face with built-in types cannot be addressed in any of the three sublanguages of OWL. The solution based on pseudo-ranges that we considered to get around the OWL limitations can be expressed within OWL-Lite. Representation of slot groupings requires OWL-full.

Based on this discussion, consider a language called OWL-Meta that has the following features:

a. Uses OWL-Lite as a representation language
b. Allows classes and slots to be instances of other classes
c. Allows *disjointWith* relationships between classes

d. Allows specialization of built-in data types

Such a language will serve the needs of the current application quite well. It builds on OWL-Lite and adds selected features from OWL-DL and OWL-full. In our earlier work [10], we surveyed an extensive range of knowledge representation systems, and OWL-Meta represents the most commonly used core across a range of systems.

Acknowledgments

We thank Richard Fikes, Ken Barker, and Bruce Porter who did the early design work for many of the representations discussed in this paper. We thank Carl Shapiro who was the original implementer of the KM-to-OWL, translator, and Sunil Mishra who subsequently maintained the translator. We thank Chris Brigham and Richard Guili for many helpful discussions on the output of the OWL translation. This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. NBCHD030010. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA or the Department of Interior-National Business Center (DOI-NBC).

References

1. Clark, P. and B. Porter. *KM -- The Knowledge Machine: Users Manual.* 1999.
2. Barker, K., B. Porter, and P. Clark, *A Library of Generic Concepts for Composing Knowledge Bases*, in *Proc. 1st Int Conf on Knowledge Capture*. 2001. p. 14--21.
3. Chaudhri, V.K., et al., *A Case Study in Engineering a Knowledge Base for an Intelligent Personal Assistant*. In the Proc. of the 2006 Semantic Desktop Workshop, Athens, GA.
4. Cheyer, A., J. Park, and R. Guili. *IRIS: Integrate, Relate, Infer, Share*. in *Semantic Desktop Workshop*. 2005. Galaway.
5. Gennari, J., et al., *The Evolution of Protege: An Environment for Knowledge-Based Systems Development. International Journal of Human-Computer Interaction*, 2003. **58**(1): p. 89-123.
6. Noy, N. and A. Rector. *Defining N-ary Relations on the Semantic Web*. 2004 [cited; Available from: http://www.w3.org/TR/swbp-n-aryRelations/.
7. Morley, D. and K. Myers. *The SPARK Agent Framework*. in *International Conference on Autonomous Agents and Multi-agent Systems*. 2004.
8. Clark, P.E., et al. *A Portable Process Language*. in *Workshop on the Role of Ontologies in Planning and Scheduling*. 2005. Monterey, CA.
9. Drummond, N., et al. *Sequences in OWL*. in *9th International Protege Conference*. 2006. Stanford, CA.
10. Chaudhri, V.K., et al., *OKBC: A Programmatic Foundation for Knowledge Base Interoperability*, in *Proceedings of the AAAI-98*. 1998: Madison, WI.