Library Harmonization Project

Use of an Ontology Editor for Library Harmonization

by John Michael Williams

Introduction

An ontology is a way of describing the way things are, in some sense. Specifically, if we have a use for such a description, an ontology can help us recognize and improve the way our understanding of some domain of knowledge represents the objective reality underlying this knowledge.

The greatest enemy of understanding is inconsistency; so, especially for large or complex bodies of knowledge, an ontology which can represent inconsistencies and isolate them for correction can be an important tool in science or engineering.

If consistent, an ontology can support logical deduction and other forms of inference leading to new insight in the domain of knowledge covered. Consistency makes learning and development of skill easier. Furthermore, consistency of meaning with terminology makes it possible to design software programs ("agents") capable of searching an ontology database interactively and drawing conclusions from its contents.

[To do: explanation of hierarchy & inheritance here; sets or classes vs attributes]

Ontology Software

In recent years, there has been work done in developing software tools to describe an ontology and to isolate deficiencies, including inconsistencies, in that ontology. The tool we shall discuss here consists of a generic user interface called Protégé, a representation system called OWL (Web Ontology Language), and a consistency checker called Racer. Recommended reading, tutorials, and example ontologies may be found at the Protégé site below; we especially recommend Horridge (2004).

All these programs are open-source freeware and may be obtained at the following locations: Protégé at http://protege.stanford.edu/, OWL at a subdirectory in the Protégé web site, and Racer at http://www.sts.tu-harburg.de/~r.f.moeller/racer/. There are precompiled binaries available for various operating systems including Windows, MacOS, Linux, and several Unix flavors. A full install may exceed 100 megabytes. For reference of the reader, the versions in use for this presentation were equal to or later than the following: Precompiled Protégé v. 2.1 (build 200), OWL v. 1.1 (build 128), and Racer v. 1.7.18,
running on a Windows 2000, 32-bit machine. Racer should be started before attempting a consistency check; it runs in background and is invoked through Protégé by system interprocess communication. Development of these programs currently (June 2004) is very active, with new Protégé builds about weekly.

**Application to the ALF Standard**

A fundamental and limiting characteristic of Protégé or OWL ontology as a representation of reality is that it depends on classification and class membership; this kind of ontology can not be applied where categories are not applicable, not known, or are not defined unambiguously. In library development, and in engineering in general, all class members or properties are artifacts, so this limitation is of no importance.

**Classes in an ALF Ontology**

To see how an ontology might be useful in library specification and development, we start with a simple example. Racer and Protégé (with OWL plugin) were started, and the Table of Contents for sections 7 and 8 of the ALF specification (IEEE Std 1603-2003) were copied over to define classes for three different ALF constructs: ALF Generic Objects (statements or declarations), ALF Library Objects (declarations only), and ALF Annotations.

The ontology "class" in OWL is not related to the "class" object in ALF; an OWL ontology "class" is closer to a set in mathematics. Following Knublauch, et al (2004), we shall use Courier typeface whenever we write the name of an OWL class.

Looking only at the Library classes, the Protégé OWL class structure that resulted is shown in Fig. 1:
The comments and annotations displayed are nonfunctional. Because all OWL classes are subclasses of Thing, the Thing class is displayed as necessary for the ALF_Library class shown selected. In other words, anything in OWL necessarily is a Thing. The ontology at this point is just a first-pass, literal copy from the ALF standard, and none of the underlying ALF relationships among the ALF_Library classes have been entered, or even thought out, at this point.

So far, an ontology appears to be nothing more than an outlining representation. Now let's see how some very basic constraints on the knowledge represented can help keep the design of the ALF specification consistent.

The classes under ALF_Library in Fig. 1 include a Wire and a Blockage. In actual cell design, wires and blockages are mutually exclusive objects. Protégé allows this relationship to be entered as a logical disjunction (mutual exclusion) on the classes of which they are members. Selecting the Wire class and entering "Blockage" in the Disjoints box shown in the lower right of Fig. 1 makes Wire and Blockage mutually exclusive, meaning that no subclass or instance in the ontology may be a joint member of both. If Blockage were selected now, "Wire" automatically would show up in the Blockage Disjoints box.

Assuming that Antenna and Blockage also are mutually exclusive, we enter "Blockage" in the Antenna Disjoints box. The result, with Blockage selected, is
shown in Fig. 2. Protégé keeps the disjunctions consistent across all affected classes, even though nothing ever was changed or edited in Blockage by the user.

![Protégé screenshot](image)

**Figure 2.** The Blockage class shows disjunctions consistent with edits made for other classes.

Notice in Fig. 2 that Blockage, which is a subclass of ALF_Library, reports that ALF_Library is necessary to it: This just means that any element of Blockage necessarily is in ALF_Library, too.

The above class structure is logically consistent, as may be tested by invoking Racer using the green [?>] button in the middle of the Protégé tool bar near the top of the figures above.

Let us now create an inconsistent class and test it for consistency, just to see how it works. Our new class will be a kind of a cell.

We select the Cell class and create a subclass called BlockWire. Then, by using the Asserted Conditions window in the middle of the figures above, we assert that both Blockage and Wire are necessary to BlockWire; this is the same as saying that BlockWire has multiple memberships and does not occupy a position in a hierarchical tree of classes. Multiple membership is not necessarily an error; but, in this case, we already have made Blockage and Wire mutually exclusive. If we now run Racer, we find that our new class BlockWire is inconsistent in the present ontology. The Racer report is shown in Fig. 3.
Figure 3. BlockWire is inconsistent because of a Disjoint entered previously for Wire, as revealed by a red outline and a Racer messagebox.

Properties in an ALF Ontology

Fixing the Meaning of our ALF_Library.

When we created the erroneous BlockWire in the example above, we said it would be a kind of cell, so we correctly made it a kind of Cell by adding it as a class under Cell. However, thinking about it, Cell isn't a kind of ALF_Library, so why is Cell under ALF_Library? Likewise, none of the classes under ALF_Library in Fig. 1, except possibly SubLibrary, is a kind of library. Something is wrong here; so, we should make a correction.

We certainly want to keep the representation in correspondence with the Std 1603 table of contents, so we shall retain three major subclasses of Thing in our ALF ontology. The easiest correction is to change ALF_Library to something different. Everything under ALF_Library in Fig. 1 is a kind of object in an ALF library; so, we decide to correct the terminology by renaming ALF_Library to ALF_LibraryObject. As a result, our naming convention becomes consistent with our ontology. This kind of consistency isn't required by Protégé, but it will help us to avoid future conceptual errors which may lead to entry errors or logical inconsistencies.
Adding Properties

The error we just have corrected was equivalent to the ontological error of representing the object classes shown as properties in the real world of an ALF_Library; whereas, these classes should have been representing real-world subclasses of a class, originally misnamed ALF_Library.

A property is a characteristic of a class other than composition of, or membership in, that class. A property represents a relationship among individuals (instances) which are members of different classes. A property may be shared by several classes; however, removing a property from a class or a class member has no effect on the identity, or count, of instances or subclasses which are members of that class. In Protégé, properties are called slots for obscure reasons; we shall use only the term property here.

Further clarification of this idea of a property may be in order: In making the correction above, renaming ALF_Library to ALF_LibraryObject, we decided to ignore libraries or kinds of them in our ontology, and to work with library objects or kinds of them, instead.

Before the correction, in the real world represented by the ontology, addition of a BlockWire had no effect on membership in ALF libraries: No matter how many of these libraries were in existence, our inconsistent BlockWire was only a new property of an ALF library.

Now, after the correction, if we added a BlockWire, we would be saying that, in the real world, we recognized an increase by one in the number of ALF library object subclasses in existence. In a different sense, BlockWire is special, though: We can not change the number of instances of ALF library objects by adding BlockWire, because, logically, BlockWire can not contain an instance, being inconsistent. Racer, not Protégé, forbids this.

Properties may be represented for any instance in a class, even if the class does not happen to include instances when the property is assigned. Properties are assigned as properties, not as classes. Protégé requires that a class and a property not have the same name. We shall adopt here the convention of naming properties by prepending "has" to the corresponding class name. Thus, when Pin is used to describe a property, it is called the hasPin property. We shall indicate the name of a property by underlining.

Returning to the corrected ALF ontology from Fig. 1, it now consists of three classes of Thing: ALF_GenericObject, ALF_LibraryObject, and Annotation. The BlockWire has been removed.

A class under ALF_LibraryObject may be associated with another class under ALF_LibraryObject to represent a property. For example, Pin may be assigned to Cell as a hasPin property. The hasPin property then may be viewed as mapping an instance in Cell to one in Pin. Likewise, Group, from ALF_GenericObject, may be assigned as hasGroup; cells typically include ALF vectors and ports, so these
properties also may be added. The assignments of the property names may be made in Protégé in the At Class window, as shown on the lower right of Fig. 4, above the Disjoints window.

![Image of Protégé window](image)

**Figure 4. Some properties of Cell are added in the At Class window.**

The new properties in Fig. 4 have no logical function, and Protégé does not automatically associate by root name (Pin and hasPin are not automatically associated), so they mean very little in the ontology at this point.

After adding the properties in Fig. 4, a form not shown here (double-click on the property) may be invoked to set the domain class for each new property to Cell and the range instance class to the corresponding root-named class. For example, the domain class of hasGroup is set to Cell, and its range instance class is set to Group. Group is a subclass of ALF_GenericObject and has not been made visible in the figures shown so far. This mapping of domain to range is how OWL properties define relationships among instances in classes. This mapping gives the properties logical function.
A Library Cell Instance in an ALF Ontology

Now, let us see how a simple ALF cell model can be represented in our ontology. The cell model, which includes only the many-to-many pin timing arc for a digital device of some kind, is as follows:

```
CELL ManyMany1
{
  GROUP AddressBit { 0 : 2 }
  GROUP DataBit    { 1 : 4 }
//
  PIN [2:0] Abus { DIRECTION = input; }
  PIN [1:4] Dbus { DIRECTION = output; }
//
  VECTOR ( 01 Abus[AddressBit] -> 01 Dbus[DataBit] )
  {
    DELAY = 1.0
    {
      FROM { PIN = Abus[AddressBit]; }
      TO   { PIN = Dbus[DataBit];    }
    }
  }
}
```

**Model 1.** ALF model of a many-to-many pin timing arc for a library cell of type ManyMany1. The cell layout and functionality are omitted.

The OWL plugin to Protégé has many configurable tabs (window arrangements); the previous figures showed only the OWL Classes tab; the slightly different Classes and Instances tab adds a window near the center of the screen for manipulating instances in an ontology. We shall use the Classes and Instances tab in subsequent figures.

**Generic Instantiation of ManyMany1**

We shall create an instance in an ontology of a completely nonfunctional cell model, just to show how it is done. The ALF constructs GROUP, PIN, and VECTOR already have been assigned as properties hasGroup, hasPin, and hasVector of a Cell in our preceding ontology, so all we need do is add a ManyMany1 subclass to Cell; the new class will inherit these properties. To instantiate a ManyMany1 type of cell, we then create a Direct Instance for each one, as shown in Fig. 5. This example shows two instances. The parenthesized "(2)" in the middle window indicates that there exist 2 instances of ManyMany1 in the ontology. The user has named these instances ManyMany1_01 and ManyMany1_02, following typical instance-naming practice in an register-transfer level (RTL) netlist. We shall use **boldfaced** typeface to indicate instances.
Figure 5. Creation of two instances of the cell class ManyMany1. The instances have been named, ManyMany1_01 and ManyMany1_02.

All At Class properties shown in Fig. 5 are inherited and thus are displayed by Protégé uncolored. The class ManyMany1 represents almost nothing of the content visible in Model 1, so we must do some more work to represent timing in our ontology; whatever we do to the class, also will be done to any instance of it.

Complete Ontology for the ManyMany1 Library Model

Studying Model 1, and recognizing that the current ALF ontology includes Group, Pin, and Vector, we shall proceed by deriving subclasses of these classes specific to ManyMany1 and then making the derived classes necessary to ManyMany1.

First, we go for the first time to ALF_GenericObject. We know that every ALF GROUP must have a domain, so we add an integer, multiple-value property, hasDomain, to the Group class there. We then create a subclass of Group called ManyMany1_Group, which will be specific to our cell. Because the Model 1 model contains two ALF GROUPs, each with different parameters, we'll further derive two classes of ManyMany1_Group, ManyMany1_AddressBit_Group and ManyMany1_DataBit_Group. By assigning Model 1 hasDomain values in these latter classes, we can instantiate them as the GROUPs AddressBit and DataBit in our cell.

The hasDomain properties will be integer types, allowed to take on multiple values, in this case, one for the left, and the other for the right, index number. After assigning the values from Model 1, the result is shown in Fig. 6.
Figure 6. Subclasses and properties of ManyMany1_Group are created for the Group contribution to the timing-arc ontology of ManyMany1.

The symbol used in the Asserted Conditions expressions is from the Protégé help menu as shown in Fig. 7.

```
<table>
<thead>
<tr>
<th>OWL Element</th>
<th>Symbol</th>
<th>Key</th>
<th>Example</th>
<th>Meaning of example</th>
</tr>
</thead>
<tbody>
<tr>
<td>allValuesFrom</td>
<td>∀</td>
<td>*</td>
<td>∀ children Male</td>
<td>All children must be of type Male</td>
</tr>
<tr>
<td>someValuesFrom</td>
<td>∃</td>
<td>?</td>
<td>3 children Lawyer</td>
<td>At least one child must be of type Lawyer</td>
</tr>
<tr>
<td>hasValue</td>
<td>∈</td>
<td>$</td>
<td>rich $ true</td>
<td>The rich property must have the value true</td>
</tr>
<tr>
<td>cardinality</td>
<td>≠</td>
<td>≥</td>
<td>children = 3</td>
<td>There must be exactly 3 children</td>
</tr>
<tr>
<td>maxCardinality</td>
<td>≤</td>
<td>&gt;</td>
<td>children ≥ 3</td>
<td>There must be at least 3 children</td>
</tr>
<tr>
<td>complementOf</td>
<td>¬</td>
<td>∈</td>
<td>Parent</td>
<td>Anything that is not of type Parent</td>
</tr>
<tr>
<td>intersectionOf</td>
<td>∩</td>
<td>&amp;</td>
<td>Human &amp; Male</td>
<td>All Humans that are Male</td>
</tr>
<tr>
<td>unionOf</td>
<td>∪</td>
<td></td>
<td></td>
<td>Doctor ∪ Lawyer</td>
</tr>
<tr>
<td>enumeration</td>
<td>{ }</td>
<td>{}</td>
<td>{male female}</td>
<td>The individuals male or female</td>
</tr>
</tbody>
</table>
```

Figure 7. A Protégé help menu lists the logical operators allowed when relating OWL properties to classes.

Having completed the Group properties, we may return to ALF_LibraryObject and similarly extend Pin. Every pin should have a direction, so we add a corresponding property to our Pin class. There happens to be an ALF_Annotation class Direction, so we shall use that class as a property range for hasDirection; we create for Direction only two instances, input and output, because that is all Model 1 requires. We can extend Direction to meet the ALF standard later, if necessary. We also add a hasPinSlice property to represent the bus indices, if any, for a pin.
With the At Class properties of a Pin defined, we create a new subclass called ManyMany1_Pin, for our model; then, for this class, we create two other subclasses, ManyMany1_Abus_Pin and ManyMany1_Dbus_Pin, to represent the two kinds of pin appearing in Model 1. We then can instantiate Abus and Dbus to denote the Model 1 pins. After associating the various property values, the result is shown in Fig. 8.

Figure 8. Subclasses and properties of ManyMany1_Pin for the timing arc ontology for Model 1. The superscript is because those classes have multiple necessary classes, in this case, ManyMany1_Pin and Direction.

There remains the most complicated statement in Model 1, the VECTOR. Our ontology only represents a timing arc, so we begin by creating just one subclass of Vector named TimingVector. The delay statement and the vector expression edge types seem to be the only unique things in Model 1, so we create the following datatype properties At Class TimingVector: hasDelay, hasToEdgeType, and hasFromEdgeType. The edgetype alternatives will be enumerations allowing just one of two strings, "01" or "10", for present purposes. We can use class references for the remainder, so we create the following object properties At Class TimingVector: hasToPin, hasFromPin, hasToPinGroup, and hasFromPinGroup.

The VECTOR in Model 1 is not named, and we do not instantiate it. The result is shown in Fig. 9.
Finally, to associate ManyMany1_TimingVector with ManyMany1, we make ManyMany1_TimingVector a necessary condition of it. We do the same with each class above that contains an instance or property necessary to define the timing arc.

Figure 9. Subclasses and properties to add ManyMany1_TimingVector.

Figure 10. The completed ManyMany1 ontology.
Our final ontology is shown in Fig. 10. The form used to make the hasGroup instance range assignment also is shown. Notice that Protégé has copied ManyMany1 as a subclass of its various necessary classes automatically.

**Closing Note**

The software currently available is beta-test quality, and its features are incompletely implemented at this writing. In general, there are limitations in quantifying class properties (quantification in the arithmetical, not formal-logical, sense) and in ordinal relations such as greater-than. Presumably, this kind of limitation will be lifted in coming months, and Protégé-based OWL will be usable in representing an EDA library.

**References**

(Available at http://protege.stanford.edu/plugins/owl/documentation.html)

Knublauth, Holger, Dameron, Olivier, and Musen, Mark A. "Weaving the Biomedical Semantic Web with the Protégé OWL Plugin".

Horridge, Matthew. A Practical Guide To Building OWL Ontologies With The Protege-OWL Plugin (v. 1.0).