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:

🖗 ALF_OWL Protégé 2.1 beta (fil	e:\D:\temp\ALF_OWL.pprj, OWL Files)	_ _ X
Project Edit Window OWL Help		
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	AR 🖸 🖻 🕸 🖻 🖡 🚟	
C)) OWLClasses		
Subclass Relationship	C ALF_Library (type=owl:Class)	+ - F T
Asserted Hierarchy 🛛 🙆 🔀 🎘	Name	💭 Annotations 🛛 💆 🥳 🛒
© owl:Thing	ALE Library	Property Value Lang
C ALF_GenericObject		D rdfs:comment 1603 Toc.8: Declaration
C Antenna	rdfs:comment	
- C Array	1603 Toc.8: Declaration	
- C Blockage		
- C Layer		
- C Node	Accented Informed	A4 Class A4 subThing All
C Pattern	Asserted mierred	At Class At own thing All
- C Pingroup	Asserted Conditions	🕫 💫 🍭 💥 📔 Properties at Class 🛛 🖸 🗜 🖳 📥 📋
- C Port	NECESS.	ARY & SUFFICIENT
C Region	O owd:Thing	NECESSARY
- C Rule	own.ming	
- C Site		
- C Vector		
- © Via		
C Wire		
C ALF_Annotation		
		🔄 🕤 Disjoints 🛛 🗹 🖓 🤹 🔍 💓
<i>d</i> 4	📥 🕸	

Figure 1. Entry of part of the IEEE Std 1603 Table of Contents into an OWL ontology.

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 editted in Blockage by the user.

PALF_OWL Protégé 2.1 b <u>eta (file</u>	::\D:\temp\ALF_OWL.pprj, OWL Files)		
Project Edit Window OWL Help			
🕒 🖨 🗿 🗠 લ 🖷 🦉 🦮 💥	AR 🖸 🖸 🔅 🤒 🖡 🖬 🔛		
OWLClasses			
Subclass Relationship	C Blockage (type=owl:Class)		+ - F T
Asserted Hierarchy 🧉 🔀 🎤	Name	Annotations	🙂 💀 🐹
C owl:Thing	Blockage	Property	Value Lang
	rdfs:comment		
- C Antenna - C Array			
C Blockage			
- Caller			
– C Node – C Pattern	Asserted Inferred	At Class At owl:T	hing All
- C Pin	Asserted Conditions	🗇 🕫 📭 🔍 💓 🛛 🎦 Properties at Cla	ass Dí Dí 🖓 🖻 📥
- O Port	NE NE	CESSARY & SUFFICIENT	
- C Primitive - C Region	C ALF Library	NECESSARY	
- C Rule			
- © SubLibrary			
- C Vector - C Via			
© Wire			
		C Antenna	
		© Wire	
#4	📥 🕸		

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.

🏶 ALF_OWL Protégé 2.1 beta (fil	e:\D:\temp\ALF_OWL.pprj,	OWL Files)	_0_
Project Edit Window OWL Help			
	(Dissidéfica de ma-swikClassa)	<u></u>	4 - 5
Asserted Hierarchy	Name) 	····
© owl:Thing			Property Value Lang
ALF_GenericObject	Mes	sage 🛛 🗶	rispeny value Lung
C Antenna	rdfs:comment	The following classes are inconsistent:	
- C Array		- BlockWire	
© Cell		ОК	
- © BlockWire M			
- O Node	Asserted Inferred		At Class At owl:Thing All
- C Pattern	Asserted Conditions	Ű Ű P Q X	PII Properties at Class D O P Properties at Class
- OP Pingroup		NECESSARY & SUFFICIENT	
- C Port	C Plockage	NECESSARY	
- C Region	Cell		
- C Rule	C Wire		
- O SubLibrary			
- C Vector - C Via			
C Wire			
C ALF_Annotation			🚽 可 Disjoints 🛛 🖞 😰 🔹 🔍 💓
#	1 🖄 🕸		

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 <u>hasPin</u> property. We shall indicate the name of a property by <u>underlining</u>.

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 assciated with another class under ALF_LibraryObject to represent a property. For example, Pin may be assigned to Cell as a <u>hasPin</u> property. The <u>hasPin</u> property then may be viewed as mapping an instance in Cell to one in Pin. Likewise, Group, from ALF_GenericObject, may be assigned as <u>hasGroup</u>; 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.

🚏 ALF_OWL Protégé 2.1 beta	(file:\D:\temp\ALF_OWL.pprj, OWL Files)
Project Edit Window OWL Help	
🕒 😅 💋 🗠 🕾 著 🏄	· 計 A R № № № № 图 目 脳 s&Instances
Subclass Relationship	Cell (type=owl:Class) + - F T
Asserted Hierarchy () () () () () () () () () () () () ()	Name Annotations Image: Cell Cell Property Value Lang Image: Cell properties added. Image: Cell properties added. Image: Cell properties added. Image: Cell properties added. Asserted Inferred At Class At Class At Class Image: Cell properties added. Asserted Inferred Image: Cell properties added. Image: Cell properties added. Image: Cell properties added. Asserted Image: Cell properties added. Image: Cell properties added. Image: Cell properties added. Asserted Image: Cell properties added. Image: Cell properties added. Image: Cell properties added. Asserted Image: Cell properties added. Image: Cell properties added. Image: Cell properties added. Image: Cell properties added. Image: Cell properties added. Image: Cell properties added. Image: Cell properties added. Image: Cell properties added. Image: Cell properties added. Image: Cell properties added. Image: Cell properties added. Image: Cell properties added. Image: Cell properties added. Image: Cell properties added. Image: Cell properties added. Image: Cell properties added. Image: Cell properties added. Image: Cell
C Region C Rule C Site C SubLibrary C Vector C Via C Wire C ALF_Annotation	C ALF_LibraryObject

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 <u>hasPin</u> 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 <u>hasGroup</u> 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
  ł
   GROUP AddressBit { 0 : 2 }
                    \{1:4\}
  GROUP DataBit
   11
  PIN [2:0] Abus { DIRECTION = input;
  PIN [1:4] Dbus { DIRECTION = output;
                                        }
  VECTOR ( 01 Abus[AddressBit] -> 01 Dbus[DataBit] )
     ł
     DELAY = 1.0
        ł
        FROM { PIN = Abus[AddressBit]; }
              { PIN = Dbus[DataBit];
        TO
     }
  }
```

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 <u>hasGroup</u>, <u>hasPin</u>, and <u>hasVector</u> 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.

ALF_OWL Protégé 2.1 beta (file:\D:\temp\ALF_OWL.pprj, OWL Files)							
Classes & Instances		10					
Relationship V C 🔄 🗙	Class	C ManyMany1 (type=owl:Class)					
C owl:Thing	© ManyMany1	Name	🚽 Annotations 🔰 🕑 🤃 🔟				
C ALF_LibraryObject ALF_LibraryObject	Display Slot	ManyMany1	Property Value Lang				
- C Array	S :NAME	rdfs:comment					
- C Blockage ♀ C Cell C ManyMany1 (2) - C Layer	Direct Instances V C h 2 X						
- C Node - C Pattern		Asserted Inferred	At Class At owl:Thing All				
- C Pin C PinGroup		Asserted Conditions 🤳 🗃 🙀	🔍 🔍 💓 Properties at Class 🗊 🗊 🗜 🖳 📥 📗				
C Port C Primitive C Region C Rule C Site C SubLibrary C Vector C Via C Wire C ALF_Annotation		NECESSARY &	SUFFICIENT NECESSARY C hasVector hasPort hasPin				
*			ා)) Disjoints 🕡 ඩු බිං 🐴 🔍 🗶				

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, <u>hasDomain</u>, 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 <u>hasDomain</u> values in these latter classes, we can instantiate them as the GROUPs AddressBit and DataBit in our cell.

The <u>hasDomain</u> 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.

ALF_OWL Protégé 2.1 beta (file:\D:\temp\ALF_OWL.pprj, OWL Fil	es)
Project cant window own, help	■ 翌
Relationship Superclass V C & X Class	C ManyMany1_AddressBit_Group (type=owl:Class)
© owl:Thing ♥ © ALF_GenericObject C ManyMany1_AddressBit_	Group M Name JAnnotations Jeiger M
C Alias Display Slot	marrywarry _AudressBil_oroup
Class	
Constant Orect Instances ∨ C ■ Orect Instances ∨ C ■ Constant	
└─ C ManyMany1_DataBit_Group ^M (1) └─ C Include	Asserted Inferred At Class At owl:Thing All
C Keyword	Asserted Conditions
C Revision	© ManyMany1_Group
└ ⓒ Template ♀- ⓒ ALF_LibraryObject	B hasDomain ⇒ 0 B hasDomain ⇒ 2 E
─ ⓒ Antenna ─ ⓒ Array	
− © Blockage • © Cell	
G ManyMany1 (2)	
	→)) Disjoints U 42 43 43 42 12

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.

OWL Element	Symbol	Key	Example Meaning of example		
allValuesFrom	A	*	∀ children Male All children must be of type Male		
some∨aluesFrom	Ξ	?	3 children Lawyer	At least one child must be of type Lawyer	
hasValue	ĥ	\$	rich ∋ true	The rich property must have the value true	
cardinality	=	=	children = 3	There must be exactly 3 children	
minCardinality	≥	>	children ≥ 3	There must be at least 3 children	
maxCardinality	м	<	children ≤ 3	3 There must be at most 3 children	
complementOf	٦	İ	⊐ Parent	Anything that is not of type Parent	
intersection Of	П	&	Human 🗖 Male	All Humans that are Male	
unionOf	Ц		Doctor ⊔ Lawyer	Anything that is either Doctor or Lawyer	
enumeration	{}	{ }	{male female}	The individuals male or female	

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 <u>hasDirection</u>; 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 <u>hasPinSlice</u> 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.

ALE OWI Protégé 2.1 beta (file:)	D:\temp\ALE_OWL_ppri_OWL_File	c)					
	AK 🕜 🗹 🦃 🖻 🖻 🖻						
Cillis Classes & Instances							
Relationship Superclass ▼ V C & ×	Class	C ManyMany1_Dbus_Pin (type=owl:Class)	C ×				
© owl:Thing	C MamMamr Dhus Pin M	Name	Annotations 📑 💀 👿 🊔				
• CALF_GenericObject		ManyMany1 Dhue Rin	Property Value Lang				
G Antonno	Display Slot	manymanyr_bbus_Pin					
C Array	P rdfs:member 🔹	rdfs:comment					
- © Blockage							
Cell Cell Cell Communication	Direct Instances V C 🖷 🗈 🗙						
- C Layer	I Dbus						
- © Node		,					
C Pattern		Asserted Inferred	At Class At owl:Thing All				
Pin ManyMany1_Pin		Asserted Conditions 🤳 🙆 😰 🌚	上 💥 🛛 Pill Properties at Class Di 🖸 🗜 🖳 📥 👘				
- C ManyMany1_Abus_Pin ^M (1)		NECESSARY & SUFFI	Dient DihasPinSlice				
C ManyMany1_Dbus_Pin ^{**} (1)		NECES	sary 0 hasDirection 9				
- © Port		A hasDirection ⇒ output					
- C Primitive		asPinSlice ∋ 1					
- C Region		(∋) hasPinSlice ⇒ 4					
- C Site							
- C SubLibrary							
- C Vector							
© Wire			🕘 Disjoints 🛛 🛈 🐢 🤹 🔍 💓				
C ALF_Annotation							
Ø4	45						
▲ ▼ •••••••••••••••••••••••••••••••••••	, aj (ara) a	I					

Figure 8. Subclasses and properties of ManyMany1_Pin for the the timing arc ontology for Model 1. The ^M superscript is because those classes have <u>multiple necessary classes</u>, 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: <u>hasDelay</u>, <u>hasToEdgeType</u>, and <u>hasFromEdgeType</u>. 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 ramainder, so we create the following object properties At Class TimingVector: <u>hasToPinGroup</u>, and <u>hasFromPinGroup</u>.

The VECTOR in Model 1 is not named, and we do not instantiate it. The result is shown in Fig. 9.

ALF_OWL Protégé 2.1 beta (file:\	D:\temp\ALF_OWL.pprj, OWL File	is)					
Project Edit Window OWL Help							
🗅 🖆 🕼 여여 🖷 🕾 著 🕅 🖉	🗅 😂 🥼 🕫 📚 著 A R 🖻 🖻 🕸 🗵 💿 🏢 🔛						
CIASSES & Instances							
Relationship Superclass ▼ V C ℑ ×	Class	C ManyMany1_TimingVector (type=owl:Class)	C X				
© owl:Thing	© ManyMany1_TimingVector M	Name	🛄 Annotations 🛛 🗳 🥵 🛒 📗				
C ALF_GenericObject ALF_LibraryObject	Display Slot	ManyMany1_TimingVector	Property Value Lang				
C Arroy	P rdfs:member	rdfs:comment					
- C Blockage							
₽- C Cell	Direct Instances 🗸 C 🐚 🌶 🗙						
🔄 🕒 🕒 🕒 🕒 🕒							
- C Layer			l				
- C Node		Asserted Inferred	At Class At authThing All				
C Pin		ASSELLEU	ACCIDIS ACOVERTING AN				
O ManyMany1 Pin		Asserted Conditions 🥑 🙆 🛓	👂 🔍 💥 📔 Pill Properties at Class 🔟 🛈 ₽ 🖳 📥 📗				
- © ManyMany1_Abus_Pin ^M (1)		NECESSARY &	SUFFICIENT D hasDelay				
— © ManyMany1_Dbus_Pin [™] (1)			NECESSARY D hasFromPin •				
- C PinGroup		O TimingVector	□ hasFromPinGroup				
C Primitive		⇒ hasDelay ⇒ 1.0 basEromEdenTune ⇒ "01"	E U hasioPin♥				
- C Region		⇒ hasFromEugeType ⇒ 01	D has To Edge Type •				
- C Rule		AddressBit	C D hasToPinGroup				
- © Site		asToEdgeType ⇒ "01"					
C SubLibrary) hasToPin ∋ Dbus					
P- C Vector		asToPinGroup ⇒ DataBit					
			- Disjoints 🗍 🖉 応 💿 🔍 🕷				
C Via							
- © Wire							
ALF_Annotation							
AA AA							
	<i>#</i> %	P					
		·					

Figure 9. Subclasses and properties to add ManyMany1_TimingVector.

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.

		🔲 hasGroup) (type=ow	l:ObjectPropert	y)			_ 🗆 🗙
								<u>c ×</u>
		Name Et	uivalent Propertie	es	📃 Ann	otations		🗾 🤹 🖾
		hasGroup				Property	Value	Lang
		rdfs:comme	nt					
ALF_OWL Protégé 2.1 beta (file:\D:\temp\ALF_OWL.	pprj, OWL Files)							
Project Edit Window OWL Help								
🗅 😅 🕼 🗠 🖙 🐂 🎥 🎥 🛣 A R 🔃 💁 🚸	og 👂 🖹 📓 🔛							
Classes & Instances		🗹 Domain de	fined	😽 Rang	le		Allows multiple	values
Relationship Superclass V C & X Class	∢ © ManyMan	Domain 🗆	÷.) 🔍 Instance		•	Inverse Function	nal
© owl:Thing	Name	Cell		Classes II	की व			
Q ALF_GenericObject	Monthlopu			Classes L	rv1 Addroce₽	it Group M	Inverse	ල් වී 🔟
C ALF_LibraryObject Display Slot				C ManyMa	ny1 DataBit G	roup ^M		
C Antenna C Array	▼ rdfs:comm						_	
- C Blockage							Symmetric	
	: 🖻 🖈 🗙 📗						Transitive	
C ManyMany1 ^M (2)								
Layer ManyMany1 02								
- C Node	Accorted	Inforrod				(At Close	At purkThing	
	Asserteu	Interreu				AL CIUSS	ACOWLITIING AI	-
O ManyMany1 Pin	Asserted	Conditions		🍈 🔨 🙀) Q.X	PII Prope	erties at Class 🔟 🚺	ŭ 🗗 🖪 🎂 📗
P O ManyMany1_Abus_Pin ^M (1)					FFICIENT	0 hasDire	ection ^O	
C ManyMany1 M (2)				NE	CESSARY 🗾	O hasToP	'in ^O	
P © ManyMany1_Dbus_Pin [™] (1)	Cell					D hasPint	Blice	
C Dia Craura	See ManyMa	any1_Abus_Pin	1 - M			0 hasToP	'inGroup ^O	
- C Port	C ManyMa	any1_AddressBr	_Group "			DhasDor	nain 🌱	
- C Primitive	C Manyin	and Dhus Pin	1			D hasGro	ugerype - un	
- C Region	C ManyMa	ny1_TimingVec	or M			O hasVec	tor	
- C Rule				IN	IHERITED	D hasPort	t	-
- C Site	hasDel	ay∋1.0	(from Many	Many1_TimingVecto	or] 💷 🎆 📗			
SubLibrary) hasDin	ection ∋ output	[from Ma	anyMany1_Dbus_Pi		Disjoin	ts 🛈 🛵 "a	ra 🙃 🔍 💓 🛛
• © TimingVector	() hasDo	noin ⇒ ?	Ifrom ManyMany	anymanyi_Abus_Pi 1 &ddroseRit Grou				
) hasDo	nain∋0	Ifrom ManyMany	1 AddressBit_Grou				
© ManyMany1 M (2)	hasDor	nain ∋ 1	(from ManyM	any1_DataBit_Grou	[] _ [q			
- © Via) 🕘 hasDoi	nain∋4	(from ManyM	any1_DataBit_Grou	p] 💶 🎬 📗			
CALE Appetation	asFro	mEdgeType ∋ "(11" (from Many	Many1_TimingVecto		1		
Arr-Armotation	hasFro	mrin ∋ Abus ∞PinGroup ⊃ Ar	[Trom Many	many1_LimingVecto				
	⇒ hasPin	HEINOIOUP ⇒ AU Blice ⇒ 4	Intesson (IIONN) (from Ma	anvManv1 Dbus Pi				
A	AL ANDIN		lfrom Mr	anuttanut Dhua Di				

Figure 10. The completed ManyMany1 ontology.

Our final ontology is shown in Fig. 10. The form used to make the <u>hasGroup</u> 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).