

# When Knowledge Models Collide

## (How it Happens and What to Do)

William E. Grosso, John H. Gennari, Ray W. Ferguson, Mark A. Musen  
Stanford Medical Informatics  
Stanford University  
Stanford, CA 94305-5479  
email: {grosso, gennari, ferguson, musen}@smi.stanford.edu

### ABSTRACT

Interoperability and reuse of components and declarative knowledge are crucial to the further development of knowledge-based software. Unfortunately, it is hard to get components to interoperate and even harder to reuse other people's work. These difficulties are often a result of incompatibilities in the knowledge models (the precise definition of declarative knowledge structures) assumed by the various components. Knowledge models are usually implicit or specified informally and arise from design choices made during system development. In this paper, we examine the choices made in Protégé, a knowledge-engineering project at Stanford University, and compare them to the choices made by the designers of the Open Knowledge Base Connectivity (OKBC) application program interface (formerly known as the Generic Frame Protocol). We show how these decisions lead to different knowledge models, discuss interoperability between Protégé and OKBC, and then close with a brief discussion of the merits of formalized knowledge models.

### 1. MOTIVATION

A casual glance through *Readings in Knowledge Representation* (Brachman and Levesque, 1985) reveals three immediate facts. The first of these is that knowledge representation is a very old branch of AI. The second is that the field has settled with age and the representational controversies of the 1970's have died down (some approaches have been abandoned and the others have learned to live with each other). The third, which is really a logical outgrowth of the second, is that the focus of the field has shifted slightly and there is more concern these days with the notions of reuse and interoperability.

This focus on reuse and interoperability naturally leads to a concern with underlying knowledge models. In order to reuse a knowledge base, we must have a good grasp of the underlying representation, both syntactically and semantically. The traditional way of explicating the underlying representation is via an embedding in some  $n$ 'th order logic (see Hayes, 1979 for an early example of doing so for a frame-based representation).

The Protégé Project is a long-running knowledge-engineering effort at Stanford SMI. It grew out of early efforts to build multi-method "shells" for knowledge-based systems. The primary focus is on providing a comprehensive toolset that enables the creation of knowledge-bases and knowledge-acquisition tools. A more recent focus is on defining and providing a set of reusable problem-solving methods and tools to construct mediators between the various components.

The Open Knowledge Base Connectivity (OKBC) API is the latest attempt to design a common API for knowledge-base management systems. It is a direct descendant of The Generic Frame Protocol (GFP) which itself grew out of Peter Karp's survey of frame representation formalisms (Karp, 1993). The primary goal is to define the semantics for a rich set of operations on knowledge-bases and thus promote interoperability.

This paper is, primarily, a discussion of the knowledge models underlying Protégé and OKBC. We hope to provide partial answers to two main questions: *why are these models different*, and *how can these differences can be ameliorated*. To answer the first question, we examine the design choices and foci of the research groups involved. To answer the second, we provide a complete axiomatization of the Protégé knowledge model (in an appendix) and discuss how the use of this axiomatization will enable an OKBC compliant server to support Protégé.

## 2. KNOWLEDGE MODELS

We use the term *knowledge model* to denote a precise, human-readable specification for a representation of declarative knowledge. That is, a knowledge model is a set of predicates and functions in some logical calculus (either first or higher order) which formally and consistently defines the meaning of every construct available within the representation.

Many representations within the knowledge-based systems community are based on the notion of “frame” (Minsky, 1985)—developers represent information using a collection of classes, instances, slots, facets, and constraints. (In many frame-based systems, facets are the only constraints. In other systems, there are no constraints whatsoever.)

By and large, frame representation systems all share the following semantics:

- A frame denotes a concept that can have attributes (slots).
- Classes and instances are represented by frames.
- Slots are also used to represent taxonomic relationships between frames (the most common relationships are *is-a*, and *part-of*)<sup>1</sup>.
- A class represents a concept rather than a specific thing.
- An instance of a class represents a specific thing (which is an instance of the concept represented by the class).
- Most slots on classes are *template slots*.

However, any attempt to be more precise than this can lead to confusion. Simply put, people mean different things when they use “class” or “instance” (see Karp, 1993, and a related discussion in Woods, 1985). For most purposes, these differences aren’t important—algorithms and theories are usually described in an informal way and intuitive descriptions suffice.

However, these differences become more important when interoperability or reuse are important. If system developers make their knowledge models explicit, then we can build translators to allow for reuse of knowledge across models. And unless these differences are made explicit, there is no way to guarantee that semantics are conserved across systems with different underlying knowledge models. To illustrate this point, and to suggest the need for precise formal specifications, we conclude this section with three examples of differences in knowledge models for frame-based representation systems: whether or not slots can be reified, the existence of meta-classes, and the role of axioms.

---

<sup>1</sup> Some authors do not think of *is-a* and *instance-of* as slots. Instead, they divide the set of binary relations into relations between classes, relations between a class and an instance, and slots.

## 2.1. Are Slots Reified?

In some systems, slots can be *reified*. That is, they are frames and can have slots themselves. This can simplify a model and make it more comprehensible. For example, an ontology might have a person class with 37 distinct slots. If these 37 slots were grouped into types (perhaps *fiscal*, *social*, *emotional*, and *physical*), then the model might be simpler and easier to use. Reified slots also make it easier to express the notion of “sub-relation.”

Whether or not slots are reified does not affect a model’s expressivity; we can always translate a reified slot into an instance of a class which has “source” and “destinations” slots. Moreover, translating between two representations, one of which allows reified slots and one of which doesn’t, is a fairly straightforward task (the general procedure is illustrated in Figure 1).

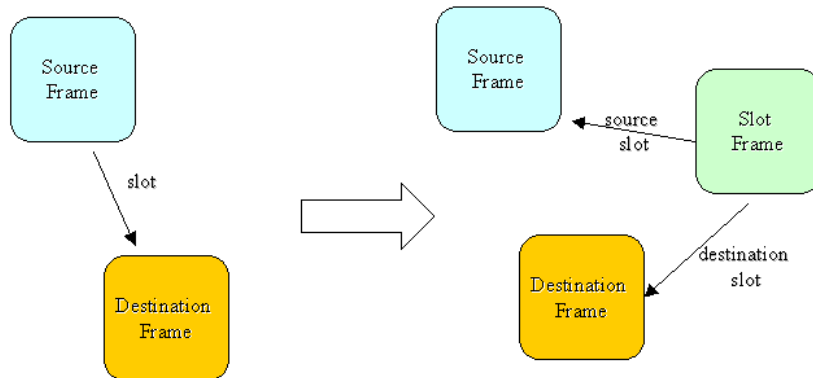


Figure 1. Reifying Slots by Hand

Presumably, reified slots are most useful when additional (e.g. beyond the set of facets supported by a particular knowledge model) constraints or extra information needs to be added to a slot . However, without either axiom schema or second order sentences, there is no uniform way to propound these additional constraints (or relate the extra information to the source frame and destination frame). Thus, without additional support from the language model, reified slots are of limited utility.

## 2.2. Are there Meta-classes?

Systems that allow meta-classes can have classes that are simultaneously instances. In such systems, it is common to distinguish two types of slots: *template* (or *member*) slots and *own* slots. Both types of slots can take values and have associated facets. The distinction lies in how they are treated when inheritance or an instance-of relation is present: template slots are inherited (as template slots) and define own slots on instances. Own slots are neither inherited nor define either type of slot for instances.

This style of modeling is natural in many domains (for an example, see the definition of the Protégé knowledge model in the appendix).

It's not entirely clear whether allowing meta-classes in a knowledge model significantly changes the underlying expressivity of the language. Certainly, the use of meta-classes allows for gradations of meaning which are not available when using a simpler representation. And, even more certainly, translating a knowledge-base which uses meta-classes into one that doesn't can be a tricky business.

### 2.3. Are Additional Axioms Allowed?

Allowing additional axioms, beyond those implied by the use of frames, to be expressed, usually expressed in some variant of KIF (Genesereth, Fikes et al, 1992), becomes more important as we try to reuse knowledge bases. Within a research group, models can be underspecified, relying on implicit assumptions that "everyone knows." But, in order to be widely reused, knowledge bases must be more complete. And even simple models like Blocks World seem to require axioms for a declarative specification (see McCarthy, 1997 for several axiomatizations of Blocks World).

For example, one physical attribute of *person* could be a *nose* slot. A perfectly reasonable starting set of axioms might be that every person has exactly one nose and that no two people share the same nose. These sentiments might be expressed as:

```
;; All people have exactly one nose
(forall ?x (=> (person ?x)
              (= (cardinality (setofall ?y (nose ?x ?y))) 1)))

;; Noses are unique
(forall (?x ?y)
  (=> (and (person ?x) (person ?y))
      (forall (?n1 ?n2) (=> (and (nose ?x ?n1) (nose ?y ?n2))
                            (/= ?n1 ?n2)))))
```

Among systems that allow axioms the most intriguing difference involves whether axioms are "enforced" (e.g. what does a system do when a knowledge base becomes inconsistent).

Axioms are usually viewed in one of 4 ways.

- *As Documentation.* In this view, theories aren't checked for consistency. Axioms serve to record the intent of the knowledge base's creators.
- *As Weak Constraints.* In this view, theories are assumed to be consistent until they can be proven false. If an instance of person is created and the nose slot does not have a value, the axiom that all people have exactly one nose has not been violated (and the theory is not inconsistent). If, on the other hand, the nose slot has two provably distinct values, then the axiom has been violated and the theory is inconsistent.
- *As Strong Constraints.* This view is similar to adopting the closed-world assumption. If an instance of person exists, and the nose slot has no declared value, then the axiom has been violated and something must be done. One strategy might be to signal an error
- *As Value-Propagation Devices.* The first three interpretations in this list concerned whether a given proposition (the axiom) is true or not. In this fourth view of the world, the axiom must hold (and the system will try to make it hold). This is much less common than the first three (what happens when the nose slot has no value? Having the system create a default nose seems a tad extreme).

Typically, a system will use some combination of these approaches. For example, a typical system might make the following decisions: Most axioms will be interpreted as weak constraints, but not checked (due to the run-time difficulties associated with undecidable algorithms); most facets will be interpreted as strong constraints; and some facets (for example, an inverse-slot facet) will be interpreted as value-propagation devices.

### **3. THE PROTÉGÉ PROJECT**

The Protégé Project is a long-running effort aimed at producing knowledge-based systems. It is primarily aimed at making it easier for domain experts to design ontologies and create knowledge acquisition tools. In addition, Protégé's long-term goals include developing a library of reusable problem-solving methods and supporting the rapid development of mediators (Gennari et al, 1994; Gennari, Grosso, and Musen, 1998). This section illustrates two of the main design choices in the Protégé toolset. We will look carefully at these decisions and illustrate their impact on the Protégé knowledge model.

#### **3.1. Full Support for Building Knowledge Bases**

There are at least three stages to the construction and use of a knowledge base: an ontology must be defined, reasonably static and long-lasting instances must be acquired, and run-time data must be entered (often when an application is running). For example, in the medical domain, these steps might correspond to gathering a medical vocabulary, acquiring specific protocols for patient care, and getting the details of a patient visit.

In many cases, it is unreasonable to expect the people building the knowledge base to be familiar with the sophisticated and subtle issues that pervade the literature on knowledge representation. Knowledge acquisition is frequently done by humans, as part of a job that requires no previous knowledge based systems experience. Protégé takes this into account and is explicitly designed to allow domain experts to build knowledge bases without constantly referring to an expert in knowledge representation.

#### **How Protégé Takes Knowledge Base Construction into Account**

The Protégé toolset takes a similar approach to that used by description logics. That is, we strongly distinguish between classes ("concepts" in description logics) and instances. This enables us to establish the following user-interface paradigm:

- Classes are concepts, related hierarchically. Protégé provides an easy to use, Windows-based ontology editor.
- Associated to each concept is a graphical prototype of a form. Protégé will automatically generate default versions of these forms which can then be customized.
- Prototype forms have user-interface controls that allow information to be easily entered and validated.
- Creating an instance involves filling out a blank form of the appropriate type. Filling out forms is a time-honored and well understood, if slightly bureaucratic, way of entering information. As such, it provides users with an easy-to-understand and easy-to-follow way of performing what often is an ancillary task for the human expert.

By including these tools inside the Protégé toolset, we have substantially lowered the barrier to knowledge base development for users, made the development of applications much simpler (since we provide almost all of the knowledge base construction code), and established a uniform look and feel across knowledge-acquisition programs.

### **Consequences for the Knowledge Model**

The above benefits come at a price. By focusing on users and on providing easy-to-use tools for designing ontologies and forms, we have deliberately chosen not to include some of the more baroque knowledge representations within our knowledge model. To do otherwise would needlessly complicate what is a fairly straightforward task in most cases. It would also have a bad effect on the user interface: when a slot is also a facet and a class, both the “class is a concept” and the “create an instance by filling out a form” metaphors break down completely.

### **3.2. Use of PSMs as Inference Engines**

The Protégé toolset also has an explicit code-reuse model, based on the idea of separating out reasoning mechanisms. Given a knowledge base, there are many different tasks that can be performed using the information in it, and many different types of inference that could be appropriate. Rather than build the inference engine into the knowledge base management system, Protégé supports the problem-solving method model, where in which reasoning is done by specialized components that can be reused in many different applications.

This model has many benefits:

- It enables easier adaptation of existing knowledge-based systems and methods of inference. Developers can use the Protégé tools for ontology design and knowledge acquisition without making any commitment to a particular PSM.
- It enables the use of various shells and libraries that have already been developed. For example, many Protégé applications include at least one CLIPS-based PSM.
- It enables collaborative development—Protégé provides a base environment and then problem-solving methods and knowledge bases developed for one application can be reused in others.
- It enables the production of lightweight knowledge acquisition tools.

### **Reuse of PSMs and Knowledge via Mapping Relations**

The question then arises: How do we reuse a problem-solving method? One possible answer is: through the use of *mediators* (Gennari, et al. 1994; Gennari, Grosso, and Musen, 1998; Park, Gennari and Musen, 1998). In the Protégé methodology, mediators define a transformation of the domain ontology into a domain-independent ontology that the problem-solving method uses (frequently referred to as the “method ontology”). The map on ontologies induces a map on the entire knowledge base and thus provides the problem-solving method with data.

### **Consequences for the Protégé Knowledge Model**

The use of mediators limits the knowledge model in much the same way that knowledge acquisition did. To see this, we once again consider the illustrative example of description logics.

Suppose that the ontologies are all expressed in one of the more tractable description-logics. And that the mediators are defined as a sequence of the following simple operations:

- Eliminating a slot
- Collapsing the distinction between a parent and a direct child if the child does not define additional slots,
- Combining two classes related by a *has-a* style relationship into a single class.
- Renaming a slot or class.
- Combining several slots using arithmetic operations (where appropriate).

Then, verifying that the mediator is sound and complete reduces to subsumption and is well-known to be decidable in polynomial time (Borgida and Patel-Schneider, 1994).

If, on the other hand, if the knowledge model is a multi-sorted second order logic, decidability is a hopeless endeavor. This is yet another instance of the classical tension between expressiveness and tractability (Levesque and Brachman, 1985): We want the knowledge model to be sufficiently expressive for good modeling and we want to be able to easily create and verify mediators that are both sound and complete. Work towards reconciling these two aims is ongoing.

#### **4. Open Knowledge Base Connectivity**

In the early 1990's, DARPA funded a series of projects aimed at reusing declarative knowledge. Among the standards developed were the Knowledge Interchange Format (KIF) and the Generic Frame Protocol (GFP). KIF was intended to be an interlingua: a way to transmit declarative knowledge between components. In this view of interoperability, knowledge-based systems would use their own representation internally, but would be able to translate their declarative knowledge into KIF statements (and create internal representations from KIF statements).

The Generic Frame Protocol (GFP) was an attempt, by researchers at SRI and Stanford KSL, to define a unified API for frame representation systems (Karp et al, 1995) and grew out of Karp's study of frame representation systems (Karp, 1993). GFP aimed at reusing procedural code. As one paper (Karp et al, 1995) put it, "Just as knowledge reuse is extremely important, so is the ability to reuse, and to mix and match, the software components of a large AI system." GFP defined a universal API for knowledge-base management systems in the hope that client-software (in particular, knowledge management utilities) would then be reusable across frame representation systems.

OKBC expands the GFP API by defining a knowledge model and behaviors, in an attempt to guarantee the semantics of commonly used API calls (Chaudhri et al, 1997). The goal of OKBC is thus explicitly to reuse knowledge bases inside a knowledge base management system (and not the KIF approach of creating an "interlingua"). The current version of OKBC explicitly incorporates KIF as an underlying representation language and thus incorporating a solid model-theoretic notion of semantics.

The goal of this section is to illustrate three of the main design choices in the OKBC specification. We will look carefully at these decisions and illustrate their impact on the OKBC knowledge model. In section 5, we will compare this knowledge model to the Protégé knowledge model.

#### 4.1. Use KIF as an Underlying Logical Language

One of the main goals of the OKBC specification is to provide a solid semantics for frame-based representations (see Hayes, 1979; Gruber, 1991; Gruber, 1993). To provide solid semantics, one must choose a logical formalism and restrict all definitions to those that can be expressed in this language. KIF was chosen as the underlying logical language for OKBC. This choice necessitates translating frame-based intuitions into logical and/or set-theoretic notions. Thus, in the history of frame-based systems, we have gone from “A frame is a data-structure for representing a stereotyped situation...” (Minsky, 1985) to “Formally, a frame corresponds to a KIF constant” (Chaudhri et al, 1997).

OKBC continues in this vein, defining *classes* to be sets (alternatively, the unary relations that define a set), *instances* to be elements of the set, *slots* to be binary relations, and *facets* to be ternary relations. Moreover, *classes*, *slots*, *instances*, and *facets* are all frames and any valid KIF sentence can be asserted about them. OKBC thus gives a very clean and precise set of logical definitions, but it’s unclear that common intuitions about frame-based knowledge representations have been adequately captured.

For example, a KIF symbol can be both a class and a slot. There is no explicit “slot” predicate in OKBC, but instead “slot-of” and “template-slot-of.” That S is a slot of F taking value V is implied by the assertion (S F V). Since KIF allows predicates to have variable arity, there is nothing to prevent S from also being a class (in the OKBC specification, a class is simply a relation with arity 1). This multiplicity of meanings can be confusing for people who have dealt with frame-based representations but not with explicit logical formulations of them.

#### 4.2. Avoid Formalisms that aren’t Widely Supported

OKBC deliberately avoids some of the more esoteric things that frame representations do. For example, it makes no commitment to either an axiom language or to any inference mechanisms within the knowledge base management server.

It’s hard to see how the OKBC designers could have done otherwise; they certainly couldn’t have legislated a universal axiom language for all frame representation systems. On the other hand, researchers developing client applications are either going to restrict themselves to a subset of the OKBC-compliant servers (partially defeating the purpose of OKBC) or avoid axiomatic representations.

#### 4.3. Use “Behaviors” to Represent System Differences

In order for a KBMS to be OKBC compliant, it must be compatible with the OKBC knowledge model. However, it need not support every nuance of the OKBC knowledge model. Behaviors are the mechanism by which a KBMS can signal that it only supports some subset of the OKBC knowledge model. The function *get-behavior* takes, as argument, the name of a behavior and returns the defined value of the behavior in the knowledge base. For example, (*get-behavior :are-frames*) might return *{:class, :instance}*. This means that neither slots nor facets are frames. Which in turn implies that neither can be either a class or an instance or have slots.

The OKBC specification defines many behaviors. Each behavior can be viewed as a set of additional axioms on top of the knowledge model. For example, the *are-frames* behavior allows a system to tell clients what gets treated as a frame (what elements of the knowledge model

correspond to KIF constants). And the return value  $\{ :class, :instance \}$  is logically equivalent to adding axioms asserting that neither slots nor facets can be frames.

Behaviors are convenient and allow for a large number of systems to quickly become OKBC compliant, at the cost of defining many different knowledge models, all of which are consistent with the general OKBC knowledge model (since all they do is add in additional constraints on what can be expressed). This is reasonable—after all, the goal of OKBC is to provide a common API for a wide variety of frame representation systems. On the other hand, it means that researchers developing client applications are either going to restrict themselves to a subset of the OKBC-compliant servers (partially defeating the purpose of OKBC) or use a fairly limited style of modeling.

## 5. COMPARING THE PROTÉGÉ AND OKBC KNOWLEDGE MODELS

This section is a description of some of the differences between the Protégé and OKBC knowledge models. As we describe in Section 6, our aim is to share knowledge across these models: to publish Protégé knowledge bases into OKBC, and to read OKBC knowledge bases into Protégé. To achieve this interoperability, we must understand all the differences between the knowledge models.

These differences can be divided into three types: *fundamental*, *structural*, and *procedural*.

- *Fundamental Differences.* Fundamental differences involve the types of constructions that are supported by the frame language (for example, whether slots can have slots) and what can be explicitly reasoned about (whether slots are reified into frames).
- *Structural Differences.* Structural differences involve the structures that the KBMS creates as side-effects of a creation operation. For example, when a frame (a knowledge-base, a class, whatever) is created, what else is created as a consequence. In many cases, a given structural difference is implied by a fundamental difference.
- *Procedural Differences.* Procedural differences involve the minor niggling details of communication with the KBMS (e.g. things like how frame handles are obtained and whether they are unique).

Of these, fundamental differences are the most interesting and come close to determining the entire knowledge model. Thus, for brevity, we have restricted this section to a table of the fundamental differences, along with explanations (Table 1). In many ways, the explanations provide capsule summaries of the discussion in previous sections.

Issue	Protégé	OKBC	Explanation
What is a frame ?	Only classes and instances can be frames	Classes, instances, slots, and facets are all frames.	Because of its focus on knowledge-acquisition, Protégé strongly enforces “slots are attributes of concepts.” Thus they have values but are not reified.
Are Meta-class Relationships allowed?	No	Yes.	This difference is rooted in both knowledge acquisition and the desire to build mediators: knowledge acquisition because end-users find the idea of meta-classes confusing (thus potentially invalidating the data) and mediators

			because of the strong “pipe/data” focus in the Protégé project.
Are there own slots on classes?	Every class is created with 3 own slots. No others can be attached in any way.	Yes. Both by inheritance (meta-classes) and by direct attachment. Users can create arbitrary own slots.	The Protégé model has evolved due to engineering requirements. One own slot, role, came about in order to distinguish between classes that need forms and classes that don’t (if no instance of the class is ever acquired, there is no need to design a form). There is also a documentation slot. And, recently, it has become obvious that a constraint slot, allowing axiomatic constraints on classes, is also necessary.
How are slots attached?	Slots can only be attached as template slots on classes. All other attachments are structural or by inheritance.	Arbitrary.	This is another example of a KA driven requirement. Allowing modelers to directly attach an own slot, while useful in some cases, may cause far more confusion than it is worth.
How are facets attached?	Each template slot has a predefined set of facets.	Arbitrary.	In Protégé, facets have a single role: they are constraints on slot values. This limits their utility but enforces a certain consistency on models (design choices appear in class definitions).
Which slots can have values?	Own slots.	Both own and template slots can have values.	
Can facet values be over-ridden in subclasses?	Yes.	No. Additional values can be asserted, but values asserted for the superclass are asserted for the subclass as well.	
Can axioms refer to anything?	Axioms can only directly refer to classes and slots.	OKBC makes no commitment to any notion of axioms (outside of a very limited tell/ask interface).	Protégé takes the point of view that an axiom which refers to an instance, or a group of instances which do not comprise a named class, is probably the result of a modeling error.

Table 1: Fundamental Differences in the Knowledge Models

## 6. MAKING PROTÉGÉ OKBC COMPLIANT

One of the fundamental truths of the knowledge-based systems field is this: knowledge is expensive. Building a knowledge-base requires domain experts, skilled ontologists, and a lot of hard work. If at all possible, we should reuse knowledge-bases, rather than rebuilding them.

This implies two things for the Protégé Project:

- (1) Tools should be developed to export knowledge-bases built in Protégé to OKBC compliant knowledge base management systems.
- (2) Tools should be developed to enable Protégé to use knowledge-bases residing in some OKBC compliant knowledge base management system.

### 6.1. Exporting to a Sufficiently Generic OKBC Server

The Protégé knowledge model is weakly compatible with the generic OKBC knowledge model. That is, we can write a set of axioms in KIF that are consistent with the OKBC knowledge model and thereby define a restricted knowledge model which is almost isomorphic to the Protégé knowledge model (as in Figure 2).

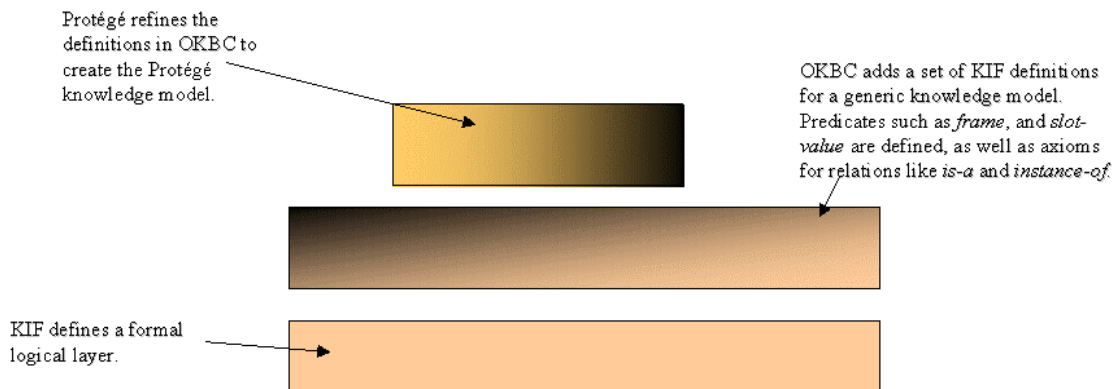


Figure 2. Viewing the Protégé Knowledge Model as a Layer on top of OKBC

For example, we have taken the statement “Every class is created with three own slots. No others can be attached in any way” and modeled it by using a distinguished meta-class as follows:

```
(class :protege-meta)
(class :protege-class)
(class :protege-thing)
(template-slot-of :protege-constraints :protege-class)
(template-slot-of :protege-documentation :protege-class)
(subclass-of :protege-class :protege-meta)
(template-slot-of :protege-role :protege-class)
(=> (subclass-of :protege-thing ?C)
    (defined-outside-protege ?C))
(=> (slot-of ?s :protege-thing)
    (template-slot-of ?y :protege-class))
(not (exists ?S (template-slot-of ?S :protege-thing)))
(instance-of :protege-thing :protege-class)
(<=> (:protege-class ?C)
    (or (= ?C :protege-thing)
        (subclass-of ?C :protege-thing)))
(=> (and (slot-of ?S ?C) (protege-class ?C))
    (template-slot-of ?S :protege-class))
```

The only known difficulty with this approach is the relationship between inheritance and template facet values. Protégé allows over-riding of template facet values and the OKBC knowledge model does not.

We plan, after testing and refining, to submit the specification for the Protégé Knowledge Model as a new behavior for OKBC. Exporting a knowledge-base built in Protégé to OKBC will then consist of the following three steps.

- (1) Tell the OKBC-compliant server to create a new knowledge base
- (2) Either the Protégé behavior flag is set (if the server supports the behavior) or the Protégé Constraint Set is asserted using the *tell* portion of the OKBC API.
- (3) The knowledge base is copied over using the OKBC API and replacing `:THING` with `:protege-thing` (a direct subclass of `:THING` with the appropriate own slots from the Protégé knowledge model) in all inheritance relationships.

## 6.2. The Problems Involved In Importing

Using a knowledge base inside an OKBC server is more problematic because we anticipate most existing knowledge bases will be made available via OKBC in roughly the same way that Protégé knowledge bases will be. That is, the tools used to create the knowledge bases will have a rather idiosyncratic knowledge model. And these idiosyncrasies will be preserved in the knowledge base available via OKBC.

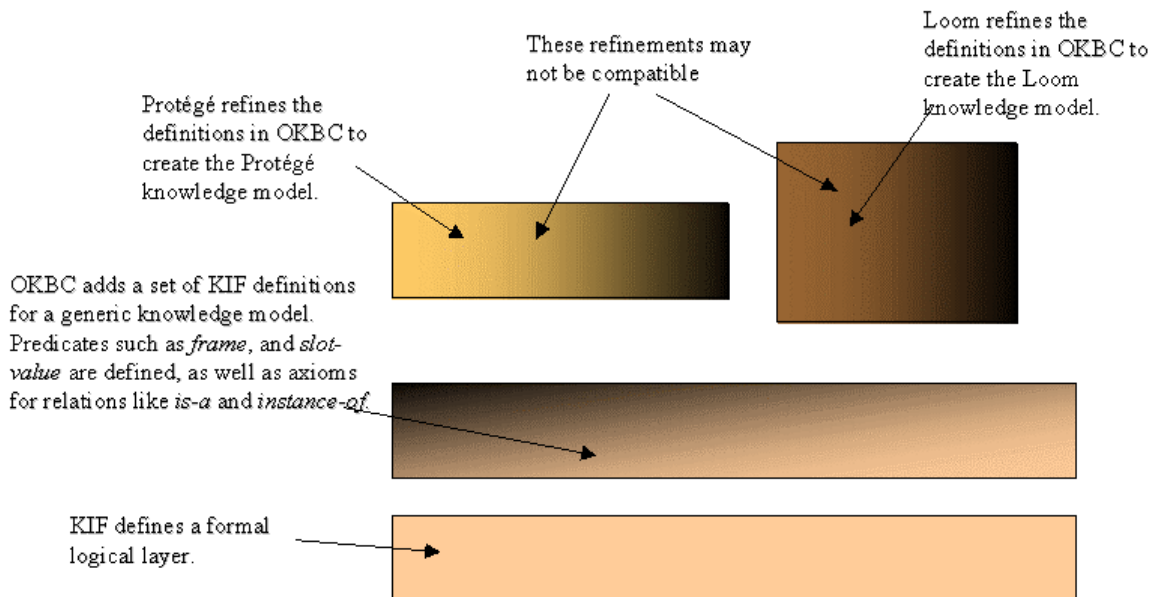


Figure 3. Reuse Across Different Knowledge Models

Figure 3 illustrates the problem: two different knowledge models, each a restriction of the OKBC knowledge model, are not necessarily compatible (and thus reusing the knowledge may be difficult).

Unfortunately, the problem is more generic than Figure 3 might indicate. A knowledge base that is perfectly valid with the original OKBC knowledge model (e.g. doesn't constrain the OKBC knowledge model in any way) may not be valid in either of the restricted knowledge models. An application which implicitly relies on one of the more restricted knowledge models will not be able to use all the information in the knowledge base.

We believe that this scenario is still a significant improvement over prior scenarios for two reasons. The first reason is that the knowledge bases are consistent with the OKBC knowledge model. While they may further restrict the meaning of some of the relations defined by OKBC, they do not contradict anything stated by the OKBC specification. Which means that large portions of the OKBC API are still quite meaningful. Our hope is that in practice, this will prove sufficient.

The second reason is that both knowledge models will have formal specifications in similar terms (e.g. the formal specifications will both incorporate OKBC's frame ontology). If the knowledge base does need to be translated, having a formal specification in a common language will undoubtedly lower the cost of doing so.

## 7. DISCUSSION

In this paper, we have discussed formal specifications for knowledge models in great detail. We believe that knowledge models are important, and will continue to grow in importance as the community continues to focus on reuse and interoperability. We base this belief on three straightforward claims which, taken together, imply that sophisticated and logically correct use of a foreign knowledge base requires knowing the underlying knowledge model:

- Formal knowledge models are very useful in pinpointing knowledge base translation difficulties and are probably the only way to reliably do so.
- Some representational constructs do not have straightforward translations. Indeed, for some, the “right” translation depends on the ontology and the intended use (e.g. there is no logically equivalent translation).
- It may be better to strive for “taxonomic equivalence” rather than “logical equivalence.” That is, the taxonomic structure of an ontology is crucial for use in applications (and for human understanding). Translators which preserve “intuitive semantics,” even at the cost of logical equivalence, may be desirable.

Our analysis of Protégé and OKBC is a start toward verifying these claims. As we described in Section 6, the ability to relate the constraints of the Protégé knowledge model to the OKBC knowledge model allows us to both formally define a mapping from Protégé to OKBC (as presented in the appendix), and to better understand the relationship between OKBC and Protégé knowledge bases.

Although our analysis does not lead to completely automatic knowledge base translation, it clarifies which constructs and semantics are problematic for interoperation. This makes the job of developing reusable knowledge bases easier— authors will know which constructs are “safe”<sup>2</sup> and which bind the knowledge base to a particular KBMS. Moreover, the analysis serves to underscore an important point— in order for a knowledge base to be truly generic, the authors need to adopt a restricted knowledge model.

If all knowledge base systems published formal knowledge models, as we have done for Protégé and as Chaudri et al. (1997) have done for OKBC, and if we can analyze and compare these

---

<sup>2</sup> A construct is “safe” to the extent that it is directly translatable into other knowledge models.

knowledge models, then we will have greatly advanced the ability to share and interoperate across knowledge base systems.

### Acknowledgements

This paper owes whatever clarity and insight it managed to achieve to the collective efforts of the (alas, anonymous) reviewers and the percipience of Samson Tu. Parts of this work were funded by the High Performance Knowledge Base Project of the Defense Advanced Research Projects Agency.

### References

- Borgida, A., Patel-Schneider P. (1994). A semantics and complete algorithm for subsumption in the CLASSIC description logic. *Journal of Artificial Intelligence Research* **1**: 277-308
- Brachman R, Levesque, H. (eds). (1985). *Readings in Knowledge Representation*. Morgan Kauffmann Publishers.
- Chaudhri, V., Farquhar, A., Fikes, R., Karp, P., and Rice, J. (1997). The Generic Frame Protocol 2.0 [Online] Available <http://www.ai.sri.com/~gfp/spec.html>.
- Davis, R., Shrobe H. , Szolovits P. (1993). What is a knowledge representation? *AI Magazine* **14**(1): 17-33.
- Finin, T., McKay, D., Fritzson, R., and McEntire, R., (1994). KQML—A language and protocol for knowledge and information exchange. In Kazuhiro Fuchi and Toshio Yokoi (Eds.), *Knowledge Building and Knowledge Sharing*, Ohmsha and IOS Press.
- Genesereth, M., Fikes, R., et al (1992). Knowledge Interchange Format 3.0 [Online] Available <http://logic.stanford.edu/kif/kif.html>.
- Gennari, J. H., Grosso, W., Musen, M. A. (1998). A method-description language: An initial ontology with examples. ). *Proceedings of the Eleventh Banff Knowledge Acquisition for Knowledge-Bases Systems Workshop*. Banff, Canada
- Gennari, J. H., Tu, S. W., Rothenfluh, T. E., and Musen, M. A. (1994). Mapping domains to methods in support of reuse. *International Journal of Human-Computer Studies*, **41**, 399–424.
- Gruber, T.R. (1991). A translation approach to portable ontology specifications. *Knowledge Acquisition*, **5**, 199-220.
- Gruber, T.R. (1993). *Ontolingua: A mechanism to support portable ontologies*. Stanford KSL Technical Report 91-66.
- Hayes, P. (1979). The Logic of Frames. In Brachman and Levesque (Eds.), *Readings in Knowledge Representation*, Morgan Kauffman, pp. 287-295.
- Karp, P, Chaudhri, V., and Paley, S. (1997) A collaborative environment for authoring large knowledge bases. submitted for publication.
- Karp, P, Myers, K., and Gruber, T. (1995) The generic frame protocol. *Proceedings of the 1995 International Joint Conference on Artificial Intelligence*, pp. 768 – 774.
- Karp, P. (1993) The design space of frame knowledge representation systems. SRI AI Center Technical Note #520.

- Levesque, H., and Brachman, R. (1984). A fundamental tradeoff in knowledge representation and reasoning (Revised Version). In Brachman and Levesque (Eds.), *Readings in Knowledge Representation*, Morgan Kauffman, pp. 41-71.
- McCarthy, J. (1997). Formulas for the Blocks World. [Online] Available <http://www-formal.stanford.edu/jmc/cs323/blocks-formulas.html>
- Minsky, M. (1981). A framework for representing knowledge. In Brachman and Levesque (Eds.), *Readings in Knowledge Representation*, Morgan Kauffman, pp. 245-263.
- Musen, M.A., Tu, S.W. (1993). Problem-solving models for generation of task-specific knowledge-acquisition tools. In J. Cuenca (Ed.), *Knowledge-Oriented Software*, Elsevier, pp. 23-50.
- Neches, Fikes, Finin, Gruber, Patil, Senator, and Swartout (1991). Enabling technology for knowledge sharing. *AI Magazine* **13**(3): 37-56.
- Newell, A. (1982). The knowledge level, *Artificial Intelligence*, **18**(2): 87-127.
- Woods, W. (1975). What's in a link: foundations for semantic networks. In Brachman and Levesque (Eds.), *Readings in Knowledge Representation*, Morgan Kauffman, pp. 217-243.

### **Appendix: A Draft Specification of the Protege Knowledge Model**

This appendix contains a draft specification for the Protege knowledge model. The specification is written in KIF and is intended to be used in conjunction with an OKBC-enabled knowledge base management system. In particular, the specification is written to simplify procedural aspects of such an implementation. For example, there is no (logical) need to define `:protege-thing` or any of the “foundational meta-classes” (see below). But doing so makes it clearer what the procedural implementation ought to do in response to various requests and, we hope, enables a fair amount of error-checking to be “on-the-fly” (we certainly hope that servers which implement this knowledge model will enforce it as well).

#### **The Axiomatization**

```
;; This is a preliminary KIF specification of the Protege knowledge model.
;; We assume standard KIF and the axiomatization of the OKBC knowledge model
;; as contained in the OKBC specification. We also assume the existence,
;; and prior definition of, the predicate defined-outside-protege

;; Notational conventions: Most symbols and relations we introduce begin with
;; ":protégé." We have adhered to the OKBC convention of using ?C, ?S, ?F,
;; ?Fa, ?I, and ?V to denote things which may be thought of, intuitively, as
;; classes, slots, frames, facets, instances, and values.

;; ***** defining the superstructure

;; There are five foundational classes. :protege-meta is an abstract
;; superclass for the other four. Each of the other four is a meta-class which
;; defines a certain set of properties for the associated Protege types.
;; In the world of the future, we might also add in :protege-axiom to allow
;; us to reason about axioms.
```

```

;; Although not logically necessary, we also define :protege-thing. The hope
;; is that converting a knowledge base is easier (and writing correct code for
;; OKBC servers is easier) if the conversion/validation process is "replace
;; :THING by :protege-thing and check to see if the resulting theory is
;; inconsistent."

(class :protege-meta)
(class :protege-class)
(class :protege-instance)
(class :protege-slot)
(class :protege-facet)
(class :protege-thing)

(subclass-of :protege-class :protege-meta)
(subclass-of :protege-instance :protege-meta)
(subclass-of :protege-slot :protege-meta)
(subclass-of :protege-facet :protege-meta)

;; We very carefully close off the meta-class hierarchy. The meta-classes
;; serve to impose some structure on a PKM-compatible knowledge base
;; (classes should be instances of :protege-class, etcetera) but should not
;; inherit from them.

(=> (subclass-of ?CSub :protege-meta)
    (or (= ?CSub :protege-class) (= ?CSub :protege-instance)
        (= ?CSub :protege-slot) (= ?CSub :protege-facet)))
(=> (subclass-of ?CSub :protege-meta) (not (exists ?F (instance-of ?CSub ?F))))

;; ***** Canonical Slots on Meta-classes
;; One of the big selling points of a meta-class hierarchy is that by defining
;; template slots on the meta-classes, we get own slots on all their
;; instances. Right now, this is 3 slots on :protege-class and 2 slots on
;; :protege-slot. If we create :protege-axiom as a metaclass (an option
;; currently left open), we will change the values of the :protege-type
;; facet for the :protege-constraints slot.

;; First, we attach the slots (strictly speaking, unnecessary—the OKBC
;; axioms infer attachment from the existence of facets or values).

(template-slot-of :protege-documentation :protege-class)
(template-slot-of :protege-documentation :protege-slot)
(template-slot-of :protege-constraints :protege-class)
(template-slot-of :protege-constraints :protege-slot)
(template-slot-of :protege-role :protege-class)

;; Now we set the facet values we know about. These facets are defined
;; further on in this specification (but their meaning is pretty clear—
;; removing the :protege and looking up the resulting predicate in the
;; OKBC spec is a good first approximation).
;; There is a slight strangeness here in that the Protégé documentation slot
;; is filled by a list of name-value pairs. We have chosen to encode each name
;; value pair as a string (in this version of the draft), rather than
;; introduce extra complexity.

(:protege-type :protege-constraints :protege-slot :STRING)
(:protege-multiple-cardinality :protege-constraints :protege-slot true)
(:protege-type :protege-documentation :protege-slot :STRING)
(:protege-multiple-cardinality :protege-documentation :protege-slot true)
(:protege-default-value :protege-documentation :protege-slot
    "Ontologist too lazy to document slot properly")

(:protege-type :protege-constraints :protege-class :STRING)
(:protege-multiple-cardinality :protege-constraints :protege-class true)
(:protege-type :protege-documentation :protege-class :STRING)

```

```

(:protege-multiple-cardinality :protege-documentation :protege-class true)
(:protege-default-value :protege-documentation :protege-class
  "Ontologist too lazy to document class properly")

(:protege-type :protege-role :protege-class :SYMBOL)
(:protege-multiple-cardinality :protege-role :protege-class false)
(:protege-allowed-values :protege-role :protege-class "abstract")
(:protege-allowed-values :protege-role :protege-class "concrete")

;; ***** Supplementing the OKBC Definitions

;; Before we can get to the meat of the Protege knowledge model, we need to
;; define some predicates.

;; In the OKBC spec, slot-of and template-slot-of are defined. The intent
;; is to axiomatize the slot role that a variable arity predicate might play
;; (rather than axiomatize the predicate as a slot). We are not so subtle
;; and we prefer to define (and then axiomatize) the notion of a
;; :protege-slot so that, later on, we can assert that everything must be
;; exactly one of a :protege-slot, :protege-facet, :protege-class, or
;; :protege-instance.

(<=> (:protege-own-slot ?S)
  (exists ?I (and (:protege-instance ?I)(slot-of ?S ?I))))
(<=> (:protege-template-slot ?S)
  (exists ?C (and (:protege-class ?C) (template-slot-of ?S ?C))))
(<=> (:protege-slot ?S)
  (or (:protege-own-slot ?S) (:protege-template-slot ?S)))

(<=> (:protege-own-facet ?Fa)
  (exists (?S ?I) (and (:protege-instance ?I) (facet-of ?Fa ?S ?I))))
(<=> (:protege-template-facet ?Fa)
  (exists (?S ?C) (and (:protege-class ?C)(template-facet-of ?Fa ?S ?C))))
(<=> (:protege-facet ?Fa)
  (or (:protege-own-facet ?Fa)(:protege-template-facet ?Fa)))

;; One simple rule to prevent inheritance loops.

(not (exists ?C (subclass-of ?C ?C)))

;; ***** defining :protege-thing and tying the superstructure in
;; :protege-thing doesn't inherit from anything inside a Protege
;; ontology, has only has the instance slots it gets from being an
;; instance of :protege-meta, and has no template slots of its own.
;; :protege-thing is logically unnecessary (we could use :protege-class
;; for the same notion) but it is a convenient, and comfortable,
;; distinction for humans to make.

(=> (subclass-of :protege-thing ?C) (defined-outside-protege ?C))
(=> (subclass-of ?C :protege-thing) (instance-of ?C :protege-class))
(=> (slot-of ?S :protege-thing)
  (template-slot-of ?S :protege-class))
(not (exists ?S (template-slot-of ?S :protege-thing)))
(instance-of :protege-thing :protege-class)

;; We now assert an exhaustive partition. This is the extremely cool
;; bit that enables the "replace :THING by :protege-thing" conversion/
;; validation heuristic to work. First we assert that a given frame
;; is an instance of at most one of the meta-classes. Then we assert
;; the partition.

(forall ?F (=> (and (subclass-of ?C1 :protege-meta) (instance-of ?F ?C1)
  (subclass-of ?C2 :protege-meta) (instance-of ?F ?C2))
  (=C1 C2)))

```

```

(forall ?F (or (defined-outside-protege ?F) (= ?F :protege-meta)
              (subclass-of ?F :protege-meta)
              (exists ?C (and (subclass-of ?C :protege-meta)
                              (instance-of ?F ?C)))))

;; ***** Classes in Protege
;; We now define more precisely what it means to be an instance of
;; :protege-class. Right now this involves three conditions: inheriting from
;; (or being) :protege-thing, only getting own slots and facets via being an
;; instance of :protege-class, and being primitive.

(<=>(:protege-class ?C)(subclass-of ?C :protege-thing)))
(=>(and (slot-of ?S ?C) (:protege-class ?C)
       (template-slot-of ?S :protege-class))
   =>(and (facet-of ?Fa ?S ?C) (:protege-class ?C)
          (template-facet-of ?Fa ?S :protege-class))
   =>(:protege-class ?C) (primitive ?C))

;; ***** Instances in Protege
;; :protege-instances are instances of :protege-class, have no template
;; slots, and have no template facets. Moreover, any slots on a protégé
;; instance come from a class that the instance is an instance of.
;; The last axiom states that if a :protege-instance is an instance
;; of some class, then that class is an instance of :protege-class

(<=>(:protege-instance ?I)
   (exists ?C (and (:protege-class ?C) (instance-of ?I ?C))))
(=>(:protege-instance ?I)
   (not (exists ?S (template-slot-of ?S ?I))))
(=>(:protege-instance ?I)
   (not (exists (?Fa ?S) (template-facet-of ?Fa ?S ?I))))
(=>(and (slot-of ?S ?I) (:protégé-instance ?I)
       (exists ?C (and (:protégé-class ?C) (template-slot-of ?S ?C) (?C ?I))))
   =>(:protege-instance ?I)
   (forall ?C (=> (instance-of ?I ?C)
                  (or (= ?C :protege-instance)
                      (instance-of ?C :protege-class)))))

;; ***** Slots in Protege
;; Template slots in Protege don't take values (values are either default
;; values, handled through a facet, or own values). When viewed as frames,
;; :protege-slots don't have template slots at all and the own slots they
;; have come from being an instance of :protege-slot

(not (exists (?S ?C ?V) (template-slot-value ?S ?C ?V)))
(not (exists (?S ?C) (and (template-slot-of ?S ?C) (:protege-slot ?C))))
(not (and (slot-of ?S ?C) (:protege-slot ?C)
          (template-slot-of ?S :protege-slot)))

;; ***** Facets in Protege
;; Our facets don't have template slots (and the own slots they do have
;; come from being an instance of :protege-facet). The last axiom states
;; that facet values cannot be set at the instance (they have to come from
;; a class facet value).

(not (exists (?S ?C) (and (template-slot-of ?S ?C) (:protege-facet ?C))))
(not (exists (?S ?C) (and (slot-of ?S ?C) (:protege-facet ?C)
                          (template-slot-of ?S :protege-facet))))
(=> (and (:protege-facet ?Fa) (:protege-instance ?I) (?Fa ?S ?I ?V))
   (exists ?C (and (:protege-class ?C)
                   (?Fa ?S ?C ?V) (instance-of ?I ?C))))

;; ***** Canonical Facets
;; Protege has only a small number of facets, predefined and with well-known

```

```

;; (to Protege users) semantics. The remainder of this specification defines
;; those facets (and asserts they are the only possible :protege-facets).
;; This list will grow—we don't currently have a consensus on the exact
;; meaning of :protege-inverse. Note also that a number of these axioms are
;; "enforcement" axioms which may or may not be useful (e.g. it depends on
;; whether the KBMS is making some form of the closed world assumption or
;; not).

(<=> (:protege-facet ?Fa)
      (or (= ?Fa :protege-type)
          (= ?Fa :protege-multiple-cardinality)
          (= ?Fa :protege-minimum-value)
          (= ?Fa :protege-maximum-value)
          (= ?Fa :protege-default-value)
          (= ?Fa :protege-allowed-values)
          (= ?Fa :protege-allowed-classes)))

;; :protege-type
;; The type facet defines the types of values that a slot can take.
;; Multiple values can be asserted for :protege-type and we use
;; the assertion :protege-class to mean that the slot can have a value
;; which is a reference to an instance.

(=> (:protege-type ?S ?F ?V)
      (or (= :protege-class ?V)(:NUMBER ?V)(:STRING ?V) (:SYMBOL ?V)))

(=> (and (holds ?S ?F ?V)(:protege-instance ?V))
      (:protege-type ?S ?F :protege-class))
(=> (and (holds ?S ?F ?V)(:NUMBER ?V))
      (:protege-type ?S ?F :NUMBER))
(=> (and (holds ?S ?F ?V)(:STRING ?V))
      (:protege-type ?S ?F :STRING))
(=> (and (holds ?S ?F ?V)(:SYMBOL ?V))
      (:protege-type ?S ?F :SYMBOL))

;; :protege-multiple-cardinality
;; The multiple-cardinality facet defines how many values a slot can take.
;; Asserting both true and false makes the theory inconsistent.

(=> (:protege-multiple-cardinality ?S ?F ?V) (or (= true ?V) (= false ?V)))
(=> (and (:protege-multiple-cardinality ?S ?F ?V1)
          (:protege-multiple-cardinality ?S ?F ?V2))
      (= ?V1 ?V2))
(=> (:protege-multiple-cardinality ?S ?F false)
      (= (and (?S ?F ?V1) (?S ?F ?V2)) (= ?V1 ?V2)))

;; :protege-minimum-value
;; A value for this facet implies two things: the slot value is of type
;; :NUMBER and the slot value is greater than the value for this facet.
;; This facet can take more than one value. :protege-maximum-value is
;; quite similar.

(=> (and (:protege-minimum-value ?S ?F ?V1) (?S ?F ?V2))
      (or (= ?V1 ?V2) (< ?V1 ?V2)))
(=> (:protege-minimum-value ?S ?F ?V) (:protege-type ?S ?F :NUMBER))

;; :protege-maximum-value

(=> (and (:protege-maximum-value ?S ?F ?V1) (?S ?F ?V2))
      (or (= ?V1 ?V2) (> ?V1 ?V2)))
(=> (:protege-minimum-value ?S ?F ?V) (:protege-type ?S ?F :NUMBER))

;; :protege-default-value
;; Defaults are strange and we do not fully axiomatize them here. The

```

```

;; basic problem is that a correct characterization requires some procedural
;; knowledge (why was a value asserted) that lies outside the domain of the
;; OKBC knowledge model.

;; In fact, there are no axioms about defaults in our system.

;; :protege-allowed-values
;; Basically, a list of allowed values for the slot. If this facet is
;; non-empty, then any values for the slot must come from this list.

(=> (?S ?F ?V1)
    (=> (exists ?V2 (:protege-allowed-values ?S ?F ?V2))
        (:protege-allowed-values ?S ?F ?V1)))

;; :protege-allowed-classes
;; A list of classes which are allowed. Recall that :protege-type could be
;; have a value of :protege-class. This slot allows an ontologist to refine
;; this further, and only allow instances of specific classes (and their
;; subclasses—we haven't chopped this off at all).

(=> (and (?S ?F ?V1)(:protege-instance ?V1))
    (=> (exists ?V2 (:protege-allowed-classes ?S ?F ?V2))
        (exists ?C1 (and (instance-of ?V1 ?C1)
                          (:protege-allowed-classes ?S ?F ?C1))))))

```