Creating Semantic Web Contents with Protégé-2000

Natalya F. Noy, Michael Sintek, Stefan Decker, Monica Crubézy, Ray W. Fergerson, and Mark A. Musen, *Stanford University*

As researchers continue to create new languages in the hope of developing a Semantic Web, they still lack consensus on a standard. The authors describe how Protégé-2000—a tool for ontology development and knowledge acquisition—can be adapted for editing models in different Semantic Web languages.

Because we can process only a tiny fraction of information available on the Web, we must turn to machines for help in processing and analyzing its contents. With current technology, machines cannot understand and interpret the meaning of the information in natural-language form, which is how most Web information is represented today. We need a Semantic Web to express information in a precise, machine-interpretable form, so software agents processing the same set of data share an understanding of what the terms describing the data mean.[1]

Consequently, we've recently seen an explosion in the number of Semantic Web languages developed. Because researchers and developers haven't yet reached a consensus on which language is the most suitable, which features each language must have, or which syntax is the most appropriate, we are likely to see even more languages emerge. We need to develop tools that will let us experiment with these new languages so we can compare their expressiveness and features, change language specifications, and select a suitable language for a specific task.

In this article, we describe Protégé-2000, a graphical tool for ontology editing and knowledge acquisition that we can adapt to enable conceptual modeling with new and evolving Semantic Web languages. Protégé-2000 lets us think about domain models at a conceptual level without having to know the syntax of the language ultimately used on the Web. We can concentrate on the concepts and relationships in the domain and the facts about them that we need to express. For example, if we are developing an ontology of wines, food, and appropriate wine–food combinations, we can focus on Bordeaux and lamb instead of markup tags and correct syntax.

Naturally, designing a new tool specifically for a new language could be better than adapting an existing tool. We can offer several reasons, however, for adapting an existing tool at the stage where no single language has emerged as the winner. First, we can experiment with emerging languages without committing enormous amounts of resources to creating tools that are custom-tailored for these languages—only to decide later that the languages are not suitable. Second, Protégé-2000 already provides considerable functionality that a new tool will need to replicate, both at the modeling and user-interface levels. Third, using different customizations of the same tool to edit ontologies in different languages gives us most of the translation among the models in the languages "for free." Translating a model from one language to another becomes as easy as selecting a "save as…" item from a menu.

Semantic Web languages

AI researchers have used ontologies for a long time to express formally a shared understanding of information. An ontology is an explicit specification of the concepts in a domain and the relations among them, which provides a formal vocabulary for information exchange. Specific instances of the concepts defined in the ontologies—*instance data*—paired with ontologies constitute the basis of the Semantic Web. In recent experiments to prototype the Semantic Web, members of different communities with different backgrounds and goals in mind have created a multitude of languages for representing ontologies and instance data on the Web (see Table1). Typically, a Semantic Web language for describing ontologies and instance data contains a hierarchical description

1094-7167/01/$10.00 © 2001 IEEE

Table 1. A selection of Semantic Web languages.

Language	Description	URL
XOL	XML-based ontology-exchange language	www.ai.sri.com/~pkarp/xol
Topic Maps	ISO standard for describing knowledge structures	www.topicmaps.org
SHOE	Simple HTML Ontology Extensions	www.cs.umd.edu/projects/plus/SHOE
RDF and RDFS	Resource Description Framework and RDF Schema	www.w3.org/RDF
DAML+OIL	DARPA Agent Markup Language + Ontology Inference Layer	www.daml.org

of important concepts in a domain (*classes*). Individuals in the domain are *instances* of these classes, and properties (*slots*) of each class describe various features and attributes of the concept. Logical statements describe relations among concepts. For example, consider an ontology describing wines, food, and appropriate wine–food combinations. Some of the classes describing this domain are **Wine**, **Wineries**, and different types of **Food**. Some properties of the **Wine** class include the wine's **flavor**, **body**, **sugar level**, and the **winery** that produced it.

These notions are present in many Semantic Web languages existing today including SHOE, Topic Maps, XOL, RDF and RDFS, and DAML+OIL.

The SHOE (Simple HTML Ontology Extensions) language, developed at the University of Maryland, introduces primitives to define ontology and instance data on Web pages. Classes are called *categories* in SHOE. Categories constitute a simple *is-a* hierarchy, and slots are binary relations. SHOE also allows relations among instances or instances and data to have any number of arguments (not just binary relations). Horn clauses express intensional definitions in SHOE.

The Hytime community developed Topic Maps, a recent ISO standard (ISO/IEC 13250). Topic Maps aim to annotate documents with conceptual information. *Topics* correspond to classes in other ontology languages and can be linked to documents. Topics are instances of *Topic Types* (other topics), which can be related to one another with *Associations*. Associations correspond closely to slots in other ontology languages. Associations belong to *Association Types*, which are again Topics. Topic Maps do not have a specialized primitive for representing instances. Any instance of a topic type can act as a topic type itself.

The bioinformatics community designed XOL for the exchange of ontologies in the field of bioinformatics. It evolved to become a general language for interchange of ontology and instance data. Being an interchange language, XOL includes primitives found in many knowledge-representation systems, object databases, and relational databases. It provides means to define classes, a class hierarchy, slots, facets, and instances.

RDF (Resource Description Framework) provides a graph-based data model, consisting of nodes and edges. Nodes correspond to objects or *resources* and the edges to properties of these objects. The labels on the nodes and edges are Uniform Resource Identifiers (URIs). However, RDF itself does not define any primitives for creating ontologies. It is the basis for several other ontology-definition languages such as RDFS and DAML+OIL.

RDF Schema[2] defines the primitives for creating ontologies. Figure 1 shows an example of a graph representing our ontology of wines as an RDFS. In RDFS, there are classes of concepts, which constitute a hierarchy with multiple inheritance. For example, the class **Wine** is a subclass of the class **Drink**. Classes typically have instances (for example, a specific red wine is an instance of the **Red Wine** class) and a resource can be an instance of more than one class (for example, **Romariz Port** is an instance of both the **Red Wine** and the **Dessert Wine** classes). Resources have properties associated with them (for example, **Wine** has **flavor**). Properties describe attributes of a resource or a relation of a resource to another resource. RDFS defines a property's *domain*—resources that can be subjects of the property—and a property's *range*—resources that can be objects of the

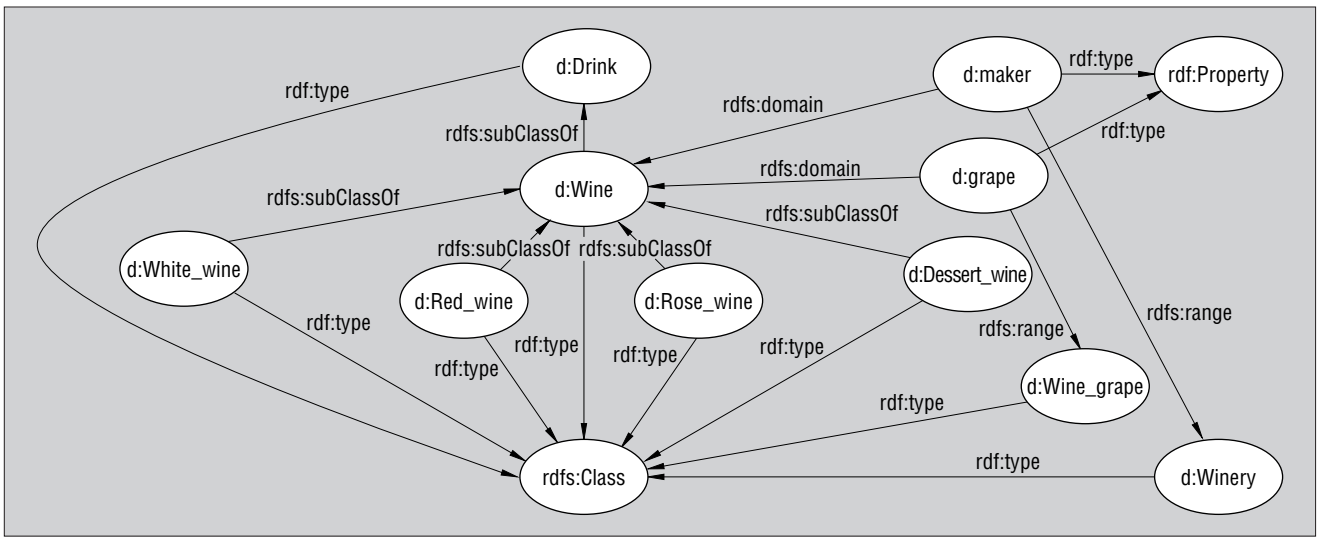


Figure 1. An RDF Schema graph representing the Wine ontology.

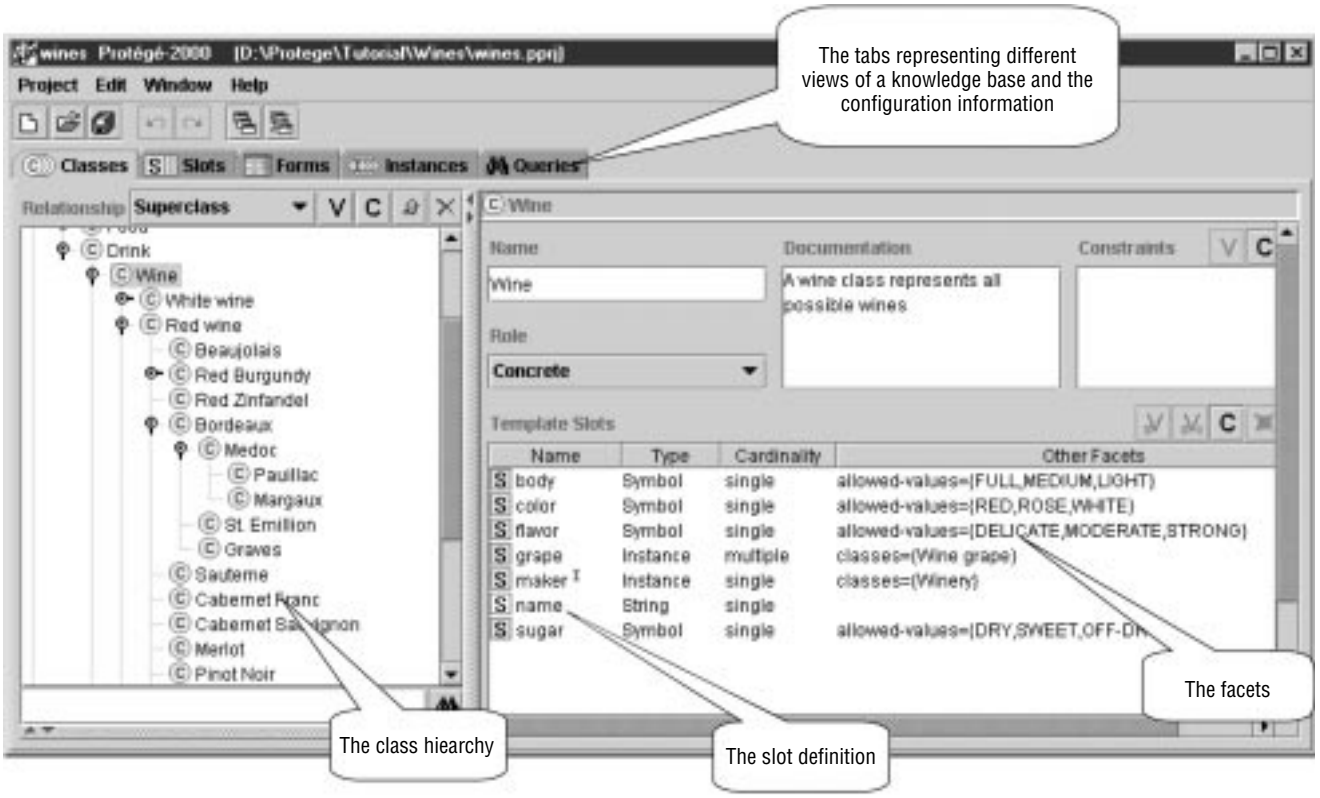


Figure 2. A snapshot of the ontology representing wines. The tree on the left represents a class hierarchy. The form on the right shows the definition of the Wine class.

property. For example, the property **maker** may have a class **Wine** as its domain and a class **Winery** as its range.

DAML+OIL (DARPA Agent Markup language + Ontology Inference Layer)[3] takes a different approach to defining classes and instances. In addition to defining classes and instances declaratively, DAML+OIL and other description-logics languages let us create *intensional* class definitions using Boolean expressions and specify necessary, or necessary and sufficient, conditions for class membership. These languages rely on inference engines (classifiers) to compute a class hierarchy and to determine class membership of instances based on the properties of classes and instances. For example, we can define a class of Bordeaux wines as "a class of wines produced by a winery in the Bordeaux region." In DAML+OIL, we can also specify global properties of classes and slots. For example, we can say that the **location** slot is *transitive*: if a winery is located in the Bordeaux region and the Bordeaux region is located in France, then the winery is in France. We will describe DAML+OIL in more detail later.

We can see from this discussion that Semantic Web languages for representing ontologies and instance data have many features in common. At the same time, there are significant differences stemming from different design goals for the languages. In adapting Protégé-2000 as an editor for these languages, we build on the similarities among them and custom-tailor the tool to account for the individual differences.

Protégé-2000

For many years now, experts in domains such as medicine and manufacturing have used Protégé-2000 for domain modeling. We show not only how we adapt Protégé-2000 to the new world of the Semantic Web—reusing its user interface, internal representation, and framework—but also how our changes enable conceptual modeling with the new Semantic Web languages.

Protégé-2000 is highly customizable, which makes its adaptation as an editor for a new language faster than creating a new editor from scratch. The following features make this customization possible:

- *An extensible knowledge model.* We can redefine declaratively the representational primitives the system uses.
- *A customizable output file format.* We can implement Protégé-2000 components that translate from the Protégé-2000 internal representation to a text representation in any formal language.
- *A customizable user interface.* We can replace Protégé-2000 user-interface components for displaying and acquiring data with new components that fit the new language better.
- *An extensible architecture that enables integration with other applications.* We can connect Protégé-2000 directly to external semantic modules, such as specific reasoning engines operating on the models in the new language.

Protégé-2000 knowledge model

Protégé-2000's representational primitives—the elements of its *knowledge model*[4]—are very similar to those of the Semantic Web languages that we described earlier. Protégé-2000 has classes, instances

of these classes, slots representing attributes of classes and instances, and facets expressing additional information about slots.

Figure 2 shows an example definition of a class, which is part of an ontology describing wines, food, and desirable wine–food combinations. In the figure, the tree on the left represents a class hierarchy. The class of Pauillac wines, for instance, is a subclass of the class of Médoc wines. In other words, each Pauillac wine is a Médoc wine. The class of Médoc wines is, in turn, a subclass of Red Bordeaux wines and so on. The form on the right in Figure 2 represents the definition of the selected class (Wine). There is the name of the class, its documentation, a list of possible constraints, and definitions of slots that the instances of this class will have. Instances of the class Wine will have slots describing their flavor, body, sugar level, the winery that produced the wine, and so on.

The form in Figure 3 displays an instance of the class Pauillac representing Château Lafite Rothschild Pauillac, and the fields display the slot values for that instance. Therefore, we know that Château Lafite Rothschild Pauillac has a full body and strong flavor among other properties. Both the class-definition forms (the right-hand side in Figure 2) and the instance-definition forms (Figure 3) are *knowledge-acquisition forms* in Protégé-2000. The fields on the knowledge-acquisition forms correspond to slot values, and we define classes and instances by filling in slot values in these fields. Protégé-2000 generates knowledge-acquisition forms automatically based on the types of the slots and restrictions on their values.

The Protégé-2000 user interface (Figure 2) consists of several *tabs* for editing different elements of a knowledge base and custom tailoring the layout of the knowledge-acquisition forms, such as the forms in Figures 2 and 3. The Classes tab defines classes and slots, and the Instances tab acquires specific instances. The Forms tab allows us to change the layout and the contents of the knowledge-acquisition forms.

We can customize almost all of the Protégé-2000 features we have described to fit a specific domain or task by

- changing declaratively the standard class and slot definitions;
- changing the content and the layout of the knowledge-acquisition forms; and
- developing plug-ins using the Protégé-2000 application-programming interface.[5]

Let's look at how we can customize Protégé-2000 and then see how we can use this flexibility to create Protégé-based editors for new Semantic Web languages.

Changing the notion of classes and slots

The definition of the Wine class in Figure 2 is a standard class definition in Protégé-2000, with a class name, documentation, list of slots, and so on. What if we need to add more attributes to a class definition, or change how a class looks, or change the default definition of a class in the system? For instance, we might want to add a list of a few best wineries for each type of wine in the hierarchy. Such a list is a property of a class (such as Pauillac wines) rather than a property of specific instances of the class (such as Château Lafite Rothschild Pauillac). The list of the best wineries for a class is not inherited by its subclasses: The best wineries producing red Bordeaux are not necessarily the same as the best Médoc or Pauillac wineries (although, they may overlap). Therefore, this list must become a part of a class definition the same way as documentation is a part of a class definition. The Protégé-2000 *metaclass* architecture lets us do just that.[4,5]

Metaclasses are templates for classes in the same way that classes are templates for instances. We can define our own metaclasses and effectively change a definition of what a class is, in the same way we would define a new class. The default Protégé-2000 template (the standard metaclass) defines the fields that we see in Figure 2. We can extend declaratively this standard definition of what a class is with new fields of any type by defining a new metaclass, which simply becomes a part of the knowledge base. Figure 4 shows a definition of the Red Bordeaux class that includes the additional field with a list of the best Bordeaux wineries.

Similarly, we can define new *metaslots* as user-defined templates for new slots. If slot definitions in our system must have fields in addition to the ones that Protégé-2000 has, we simply define new templates where we describe these new fields.

Custom-tailoring slot widgets for value acquisition

The look and behavior of the fields on the knowledge-acquisition forms in Figures 2 and 3 depend on the types of the values that the fields can take. A field for a string value, such as a class name, has a simple text window. A field that contains a list of complex values is a list box with buttons to add and remove values and to create new values. These fields are called *slot widgets*. They not only display the values appropriately, but also help to ensure that the values are correct based on the slot definitions in the ontology. For example, the maker of a wine must be a winery—an instance of the Winery class. The slot widget for the maker slot in Figure 3 lets us set the value only to a Winery instance.

Developers can extend Protégé-2000 by implementing their own slot widgets that are tailored to acquire and verify particular kinds of values. Suppose we wanted to be more precise about the sugar level in wine and to mark it on a scale rather than simply choosing among three values—dry, sweet, or off-dry.

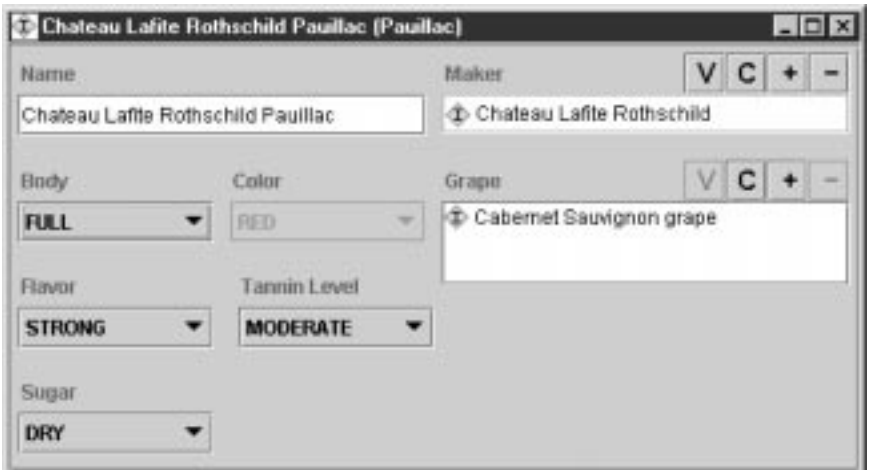


Figure 3. An instance of the class Pauillac representing the Château Lafite Rothschild Pauillac. This wine has a full body, a strong flavor, and a moderate tannin level, among other properties.

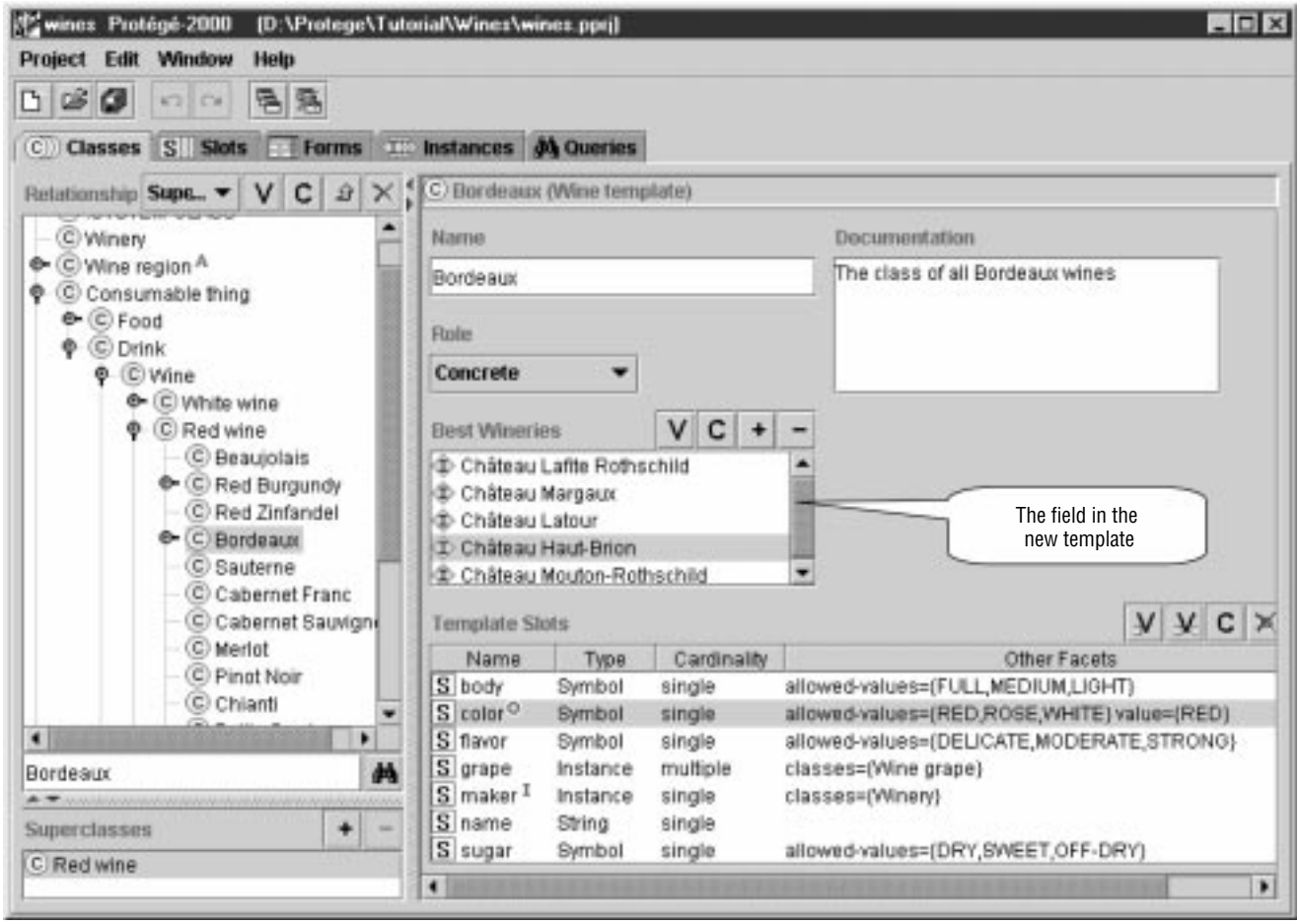


Figure 4. A class definition that uses a nonstandard template. We added the best wineries slot to the standard class-definition template.

We could store the value as a number in the sugar slot. We could use a slot widget that would let us select the value on a slider rather than enter a number (see Figure 5). When we customize knowledge-acquisition forms, we choose not only the layout of the fields on the form, but also the slot widgets that must be used for different fields. The slot widgets we choose do not usually affect the contents of the knowledge base itself, but their use can make the look and feel of the tool much more suitable for a particular domain or language. Slot widgets also can help ensure the internal consistency of a knowledge base by checking, for example, that an integer value that we enter is between the allowed maximum and minimum for that slot.

Using a back-end plug-in to generate the right output

When we develop a domain model in Protégé-2000, we do not need to think about the specific file format that Protégé-2000 will use to save our work. We think about our domain at the conceptual level and create classes, slots, facets, and instances using the graphical user interface. Protégé-2000 then saves the resulting domain models in its own file format. Developers can change this file format in the same way they plug in slot widgets. Back-end plug-ins let developers substitute the Protégé-2000 text file format with any other file format. For example, suppose we wanted to use XML to publish the wine ontology and other domain models we create using Protégé-2000. We would then need to create an XML back end that substitutes files in the Protégé-2000 format with the files in XML. A back end creates a mapping between the in-memory representation of a Protégé-2000 knowledge base and the file output in the required format. The back end also enables us to import the files in that format into Protégé-2000. The new file format has the same status as the Protégé-2000 native file format, and the users can choose either format for their files.

Editing Semantic Web languages with Protégé-2000

Armed with the arsenal of tools to custom-tailor Protégé-2000 quickly and easily, let's look at what is involved in creating a Protégé-2000 editor for a new Semantic Web language. We will use the Protégé-RDFS editor developed in our laboratory as an example, but the ideas are the same for any new language.

We start creating a Protégé-2000 editor for our new language by determining the differences between the knowledge models of the two languages: the knowledge model of Protégé-2000 and the knowledge model underlying our language of choice. We then decide which of the available tools—metaclasses, custom user-interface components, or a custom back end—we will use to reconcile the differences or to hide them from the user.

In practice, the overlap between the knowledge models underlying the Semantic Web languages available today is very large. The models might use different terminology for

the same notion (for example, *slots* in Protégé-2000 and *properties* in RDFS). However, the structure of the concepts, the underlying semantics, and the restrictions are often similar.

When we compare the two knowledge models, we identify four categories of concepts (see Figure 6):

1. *Concepts that are exactly the same in the two languages (possibly with different names)*. Usually, classes, inheritance, instances, slots as properties of classes and instances, and many of the slot restrictions fall into this category.
2. *Concepts that are the same but expressed differently in the two languages*. For example, Protégé-2000 associates slots with classes by attaching a slot to a class. RDFS defines essentially the same relationship by defining a domain of a property.
3. *Concepts in our language of choice that do not have an equivalent in Protégé-2000*. For example, RDFS allows an instance to have more than one type, whereas in Protégé-2000 each instance has a unique direct type.
4. *Concepts that Protégé-2000 supports and our language of choice does not*. For example, Protégé-2000 allows a slot to have more than one allowed class for its values, whereas the range of a property in RDFS is limited to a single class.

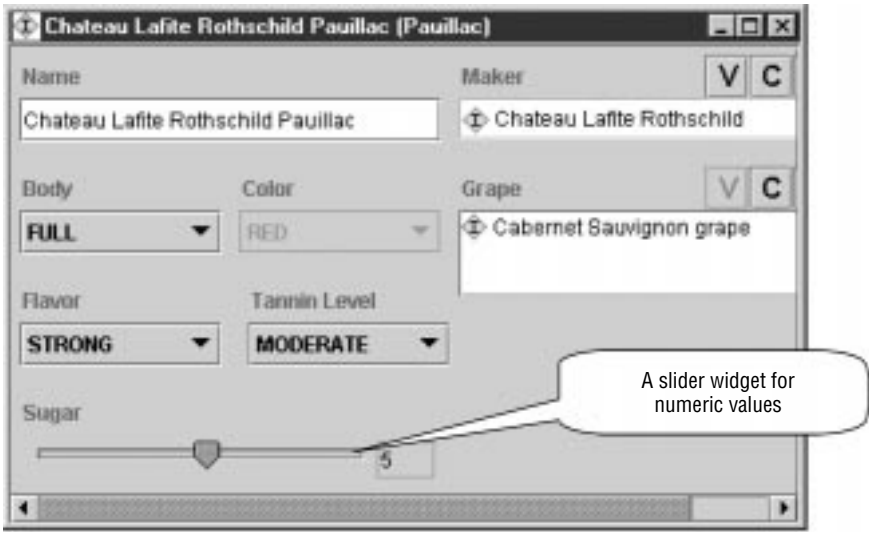
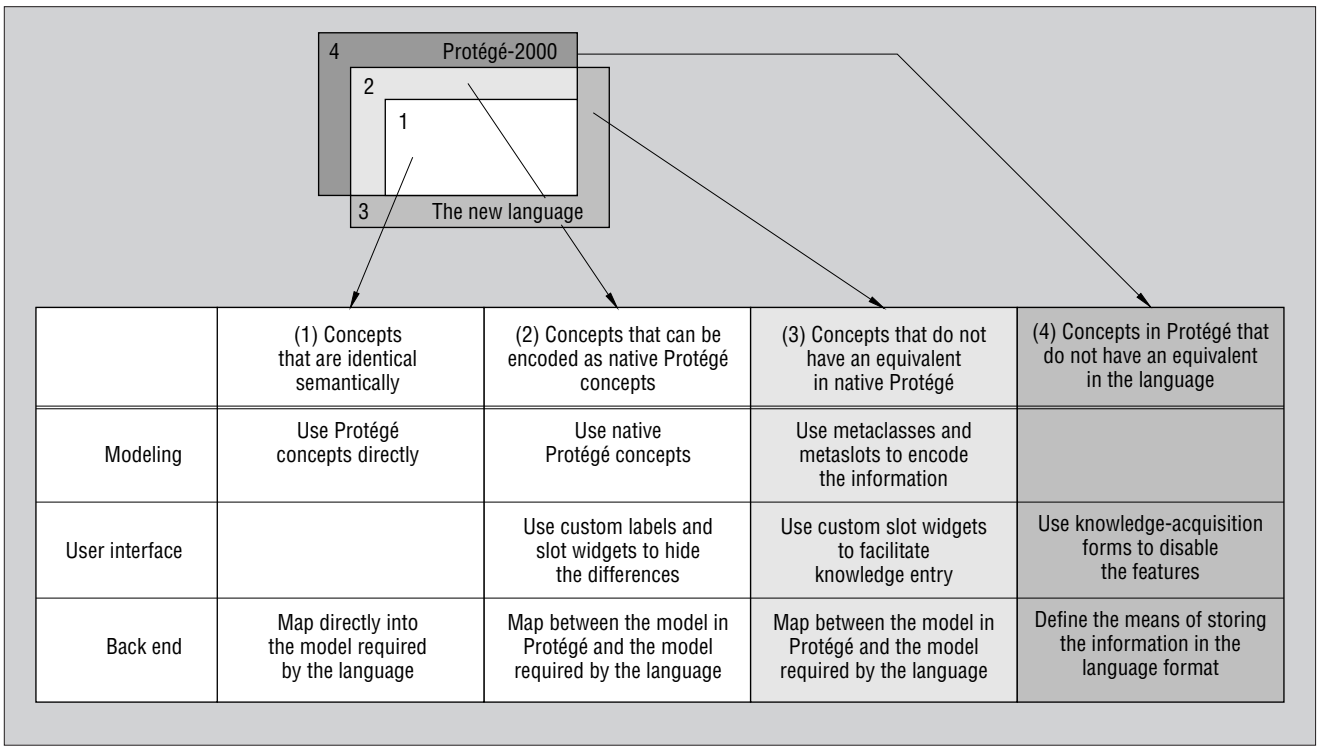


Figure 5. Changing a slot widget. We use a slider instead of a simple field to acquire numeric values for the sugar level.

Naturally, we can express all the features of our language that fall into the first category directly in Protégé-2000. We deal with the differences in the other three categories by defining appropriate metaclasses and metaslots and by resolving the remaining changes in the back end. We hide the differences from the user behind custom-tailored slot widgets.

The second item on the list, the concepts that do not have a direct equivalent in Protégé-2000 but that can be mapped to native Protégé-2000 concepts, deserves a special discussion. Consider domains of properties in RDFS (rdfs:domain). A domain specifies a class on which a property might be used. For example, the domain of the flavor property is the Wine class. Protégé-2000 slots are similar to properties in RDFS. Attaching a slot to a class in Protégé-2000 also specifies that a slot can be used with that class. For example, the flavor slot



	(1) Concepts that are identical semantically	(2) Concepts that can be encoded as native Protégé concepts	(3) Concepts that do not have an equivalent in native Protégé	(4) Concepts in Protégé that do not have an equivalent in the language
Modeling	Use Protégé concepts directly	Use native Protégé concepts	Use metaclasses and metaslots to encode the information	
User interface		Use custom labels and slot widgets to hide the differences	Use custom slot widgets to facilitate knowledge entry	Use knowledge-acquisition forms to disable the features
Back end	Map directly into the model required by the language	Map between the model in Protégé and the model required by the language	Map between the model in Protégé and the model required by the language	Define the means of storing the information in the language format

Figure 6. Comparing the knowledge models of Protégé-2000 and a new Semantic Web language.

Table 2. Creating the Protégé-based RDFS editor.

Category	(1) Concepts in RDFS that are (nearly) identical to Protégé concepts	(2) Concepts in RDFS that can be encoded as native Protégé concepts	(3) Concepts in RDFS that do not have an equivalent in native Protégé	(4) Concepts in Protégé that do not have an equivalent in RDFS
Modeling	rdfs:Class = :STANDARD-CLASS rdfs:subClassOf = subclass of rdf:type = instance of rdf:Property = :STANDARD-SLOT rdfs:subPropertyOf = subslot of rdfs:Resource = :THING rdf:comment = :DOCUMENTATION	Do not use explicit rdfs:domain and rdfs:range for properties; rdfs: domain encoded as slot attachment; rdfs:range encoded as allowed class Instance-typed slots	rdfs:Class is a default for metaclass, and rdf:Property is a default metaslot; add properties rdfs:isDefinedBy, rdfs:seeAlso as core slots; add rdfs:ConstraintProperty and rdfs:ConstraintResource as core classes; multiple types of an instance	Cardinality, inverse slot, and default value facets; multiple allowed classes for a slot
User interface	Custom labels on class and slots forms (for example, "Properties" and "Comment")		Plug-in URI slot widget for validating URI-type slots.	Disable default-value and inverse-slot widgets on slot forms.
Back end	Map Protégé concepts directly to RDFS concepts	Translate slot attachments as rdfs:domain for properties and allowed classes as rdfs:range	On import, create new class as a subclass of the multiple types.	Write out extra facet information as Protégé-specific properties on properties. If a slot has multiple allowed classes, create a new class for rdfs:range value. On import, do the reverse.

is attached to the **Wine** class. We have two ways to encode the RDFS domain information in a Protégé-RDFS editor. First, we can add a **domain** slot to a template (metaslot) that we will use for all our slots. Then, a field for **domain** will appear on each form for a slot, and we will fill it in there. Second, we can simply use the native Protégé-2000 notion of slot attachment and translate the attachments of slots to classes into domains of properties in the back end. The second solution lets us use the Protégé-2000 user interface directly and hides the features of a specific language used to store the information.

We find it extremely beneficial to adopt the paradigm of using the native Protégé-2000 features wherever possible and of resorting to additional definitions, such as metaclasses and metaslots, only when absolutely necessary. This approach maximally facilitates the exchange of domain models among different languages, which we edit (or will edit) with Protégé-2000. As new languages emerge and we experiment with them, the knowledge models underlying these languages will undoubtedly overlap. By encoding as much as possible in the native Protégé-2000 structures and leaving part of the translation between the Protégé-2000 model and the language to the back end, we maximize the amount of information that we will preserve by simply loading a knowledge base in one language supported by Protégé-2000 and saving it to another language. Even though there would often be some parts of these models that will not be part of this overlap, we are maximizing the amount of information that gets ported among models in different languages for free.

Having generated the four groups of concepts after comparing the two knowledge models (see Figure 6), we can reconcile the differences using

1. modeling—by changing default definitions of classes and slots at the modeling level;
2. the user interface—by developing specialized user-interface components; and
3. the back end—by implementing the new back end that will translate between the domain model in Protégé-2000 and the domain model in our language of choice.

Let's look at how each of these three levels works using the development of a Protégé-based RDFS editor as an example (see Table 2 for a summary of the entire process).

The modeling level

We start by determining which concepts in our language of choice are identical to Protégé-2000 concepts or that can be represented using the native Protégé-2000 concepts. We use the native Protégé-2000 as a means to model this group of concepts, even if it is not how these elements are directly expressed in our language. We then define the new templates for class and slot definitions if necessary.

Consider, for example, the two attributes—**rdfs:seeAlso** and **rdfs:isDefinedBy**—that are associated with each class and each property in RDFS. The **rdfs:seeAlso** property specifies another resource containing information about the subject resource, and the **rdfs:isDefinedBy** property indicates the resource defining the subject resource. The values of these properties are other resources or URIs pointing to other resources. We must add these two fields that the Protégé-2000 itself does not have to each class and slot form in our knowledge base. To add these fields, we define a new metaclass that will serve as a template for RDFS classes. This metaclass is, in fact, equivalent to the RDFS class **rdfs:Class**. Figure 7 shows the definition of **rdfs:Class** with the new template slots that will appear on each class form that uses this template.

The user-interface level

When creating a Protégé-based editor for a new language, we can change both the behavior and the look and feel of the knowledge-acquisition forms to reflect the terminology and the features of the language. First, we can change the labels on the forms—the simplest type of customization. For example, we can easily replace Protégé's "Template slots" label in a class definition with the RDFS "Properties" label to give the form an RDFS look. Other elements that we can easily configure by manipulating the forms include whether or not a field should be visible to the

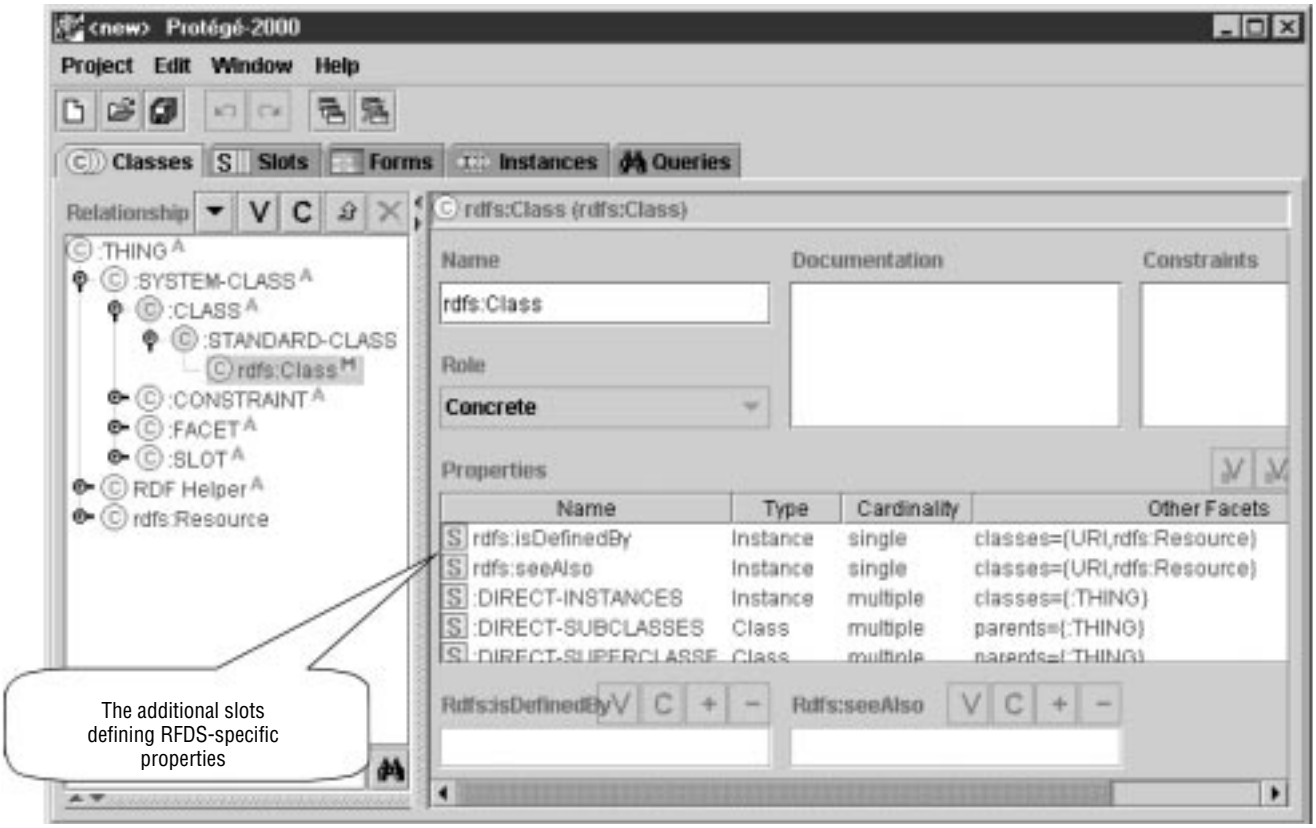


Figure 7. A template definition for classes in RDFS. The class rdfs:Class inherits most of the slots from the standard class template, but the two slots at the top of the list of properties are the ones that we defined for RDFS.

user, the buttons on the fields, the position and size of the fields, and the slot widgets to be used for each field. We perform this configuration entirely in the Protégé-2000 Forms tab and not in the programming code.

We could also develop our own slot-widget plug-ins to allow editing and verification of elements that are unique to our language. For example, a URI widget in the Protégé-RDF editor can validate that the user has entered a correct URI or even take the user to the corresponding Web page.

Disabling fields for some slots on the form prevents the user from exercising Protégé-2000 features that the particular Semantic Web language does not support. For example, we can disable the field for entering default slot values in the Protégé-RDFS editor, because RDFS does not support default values.

The back-end level

Whatever the differences between Protégé-2000 and our language that we could not resolve at the modeling and user-interface levels, we will need to reconcile in the module that saves the internal Protégé-2000 representation in the required output file format—the back-end plug-in. The back-end plug-in

1. saves a Protégé-2000 knowledge base in a file format that conforms to the syntax of our language of choice;
2. maps the elements of the Protégé-2000 knowledge base that do not have a direct equivalent in our language to the appropriate set of elements in this language; and
3. imports domain models in this language that were developed elsewhere for editing in Protégé-2000.

Usually, when developers define a language with a new syntax, they quickly implement a parser that allows developers to read and write files in that language's syntax. Many of the new languages are extensions of XML or RDF, and thus we can often use the existing XML and RDF parsers to take care of the syntactic part of adapting to the new language.

In RDFS, the back end must deal with a number of issues that we did not resolve at the modeling level or in the user interface. We might have resolved some of these issues there, but it would have unnecessarily complicated the editor for the user. For example, instances in Protégé-2000 are of a single class, whereas in RDFS they can be direct instances of several classes (for example, they have several direct types). Because the RDFS model is more general, we have no problem in saving a Protégé-2000 knowledge base in RDFS. However, when we import RDFS instance data into Protégé-2000, we must deal with instances that have several direct types. Suppose we have a class for red wines and a class for dessert wines. We have Romariz Port as an instance of both classes in RDFS. When we import this RDFS instance data into Protégé-2000, the back end can create a new class that is a subclass of both classes (for example, denoting a concept of dessert red wines) and make the Romariz Port instance an instance of this new class. We can record the two original classes of Romariz Port as additional slots on the newly created class (as shown in Figure 4). When saving back to RDFS, the back end can extract the information from this slot, thus preserving the original model.

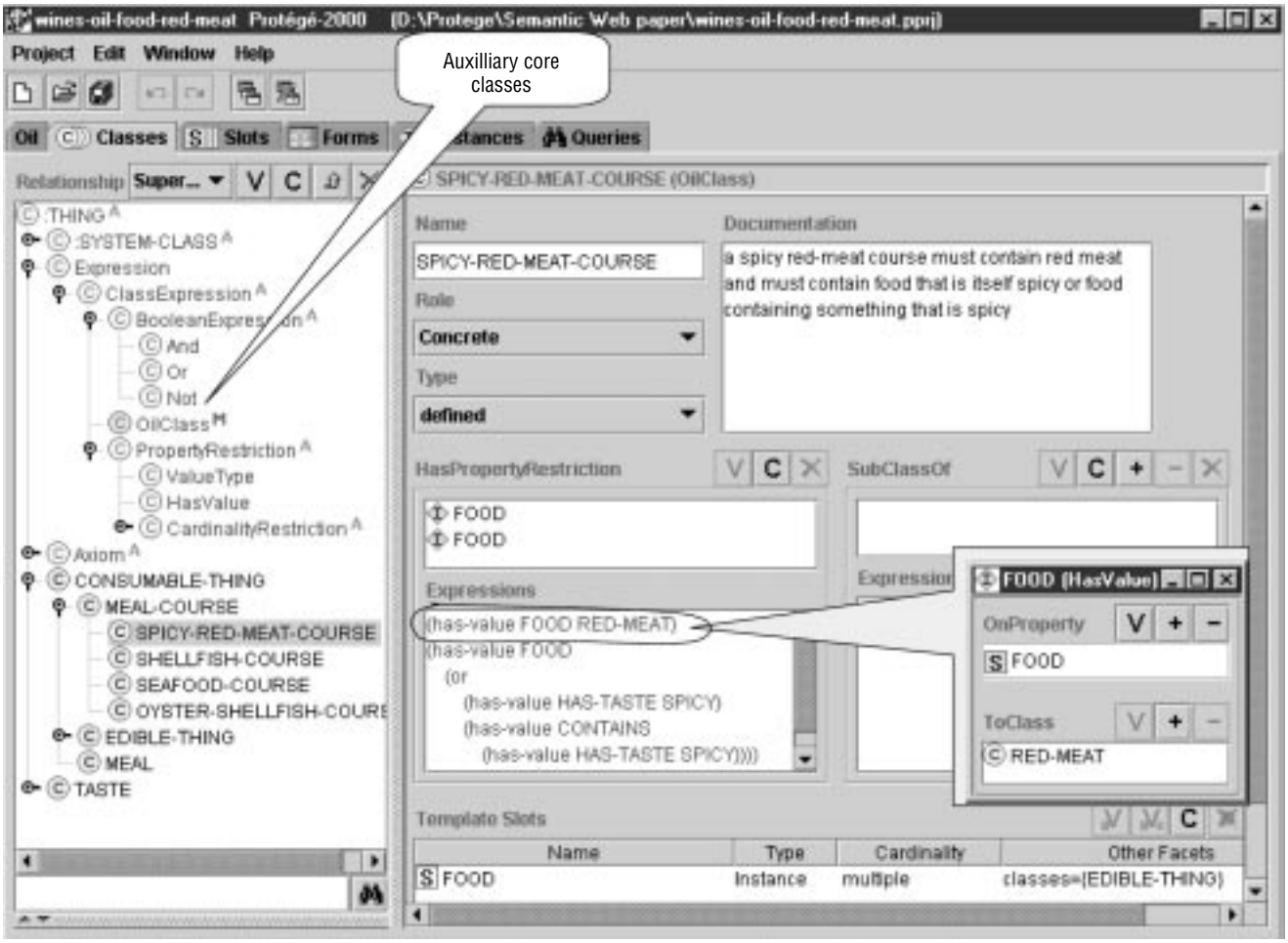


Figure 8. The definition for the spicy-red-meat-course class in the Protégé-OIL editor. In addition to the standard fields, such as those shown in Figure 2), we have OIL-specific fields such as hasPropertyRestriction and subClassOf for specifying complex class expressions. These slots use the OIL-specific slot widget to display expressions. The tree on the left contains the auxiliary core classes we defined for OIL.

Any user-defined back end has the same status as all the other back ends, including the ones that are part of the core Protégé-2000 system: it can be used as a storage format for Protégé-2000 knowledge bases. Therefore, there is another, no less important, goal of a back-end plug-in: to ensure that when we create a knowledge base in Protégé-2000, save it using the back-end plug-in, and load it again, we have preserved all the information. Hence, we must find a way to store the elements that Protégé-2000 supports, but that our language of choice does not. Most Semantic Web languages are flexible enough to easily store this information. For example, in RDFS, we simply add new Protégé-specific properties to slots to record default values, which RDFS does not have. These properties have no meaning to another RDFS agent, but if we read the knowledge base back in Protégé-2000, we will have the default values preserved.

Creating new tabs to include other semantic modules

In addition to creating a Protégé-based editor for a new Semantic Web language, developers can plug in other applications in the knowledge-base–editing environment. In addition to the standard *tabs* that constitute the Protégé-2000 user interface (Figures 2 and 4), developers can create *tab plug-ins* in the same way they can plug in new slot widgets. These tabs can include arbitrary applications that benefit from the live connection to the knowledge base. These applications then become an integral part of the knowledge-base–editing environment.

Consider our wine example again. Having created a knowledge base of wines and food and the appropriate combinations, we might want to build an application that produces wine suggestions for a meal course in a restaurant. Such an application would actively use the data in the knowledge base but it would also implement its own reasoning mechanism to analyze suggestions. We can implement this wine-selection application as a tab plug-in.

In practical terms, a tab plug-in is a separate application, a developer's own user-interface space from which the developer can connect to, query, and update the current Protégé-2000 knowledge base.

In the realm of the Semantic Web, a tab can include any applications that would help us acquire or annotate the knowledge base. Such applications can

- enable direct annotation of HTML pages with semantic elements;

- provide connection to external reasoning and inference resources;
- acquire the semantic data automatically from text; and
- present a graphical view of a set of interrelated resources.

Using Protégé to edit DAML+OIL

DAML+OIL, the Semantic Web language that was heavily inspired by research in description logics (DL), allows more types of concept definitions in ontologies than Protégé-2000 and RDFS do. The DL-inspired languages usually include the following features in addition to the ones found in the traditional frame-based languages:

- We can specify not only *necessary* but also *sufficient* conditions for class membership. For example, if a wine is produced by a winery from the Bordeaux region, it is a Bordeaux wine.
- We can use arbitrary Boolean expressions in class and slot definitions to specify superclasses of a class, domain and range of a slot, and so on. For example, a spicy red-meat course must contain red meat and must contain food that is itself spicy *or* food containing something that is spicy.
- We can specify global slot properties. For example, location is a *transitive* property: if the Château Lafite Rothschild winery is in the Bordeaux region and the Bordeaux region is in France, then the Château Lafite Rothschild winery is in France.
- We can define global axioms that express additional properties of classes. For example, we can say that the classification of the class of all wines into the subclasses for red, white, and rosé wines is *disjoint*: Each instance of the Wine class belongs only to one of these subclasses.

We have adapted Protégé-2000 to work as an OIL editor. (The OIL language is a precursor for DAML+OIL.) In doing so, we followed the same steps we described in creating the Protégé-based RDFS editor. In addition, we have integrated external services for OIL ontologies into Protégé-2000. Integrating DAML+OIL would require nearly identical steps.

The modeling level

We introduce the new class and slot templates, OilClass and OilProperty, to specify complex class and slot definitions. As a result, a class template, for example, acquires these three new fields (see Figure 8):

1. type—to specify whether the class definition contains only necessary or both necessary and sufficient conditions for class membership;
2. hasPropertyRestriction—to specify complex expressions for slot restrictions; and
3. subclassOf—to specify complex expressions describing the position of the class in the class hierarchy.

To integrate OIL into Protégé-2000, we used the names from the RDFS serialization syntax of (Standard-)OIL and not the plain ASCII version. See www.ontoknowledge.org/oil/ for the various syntaxes and versions.

Just as for RDFS, we use as many native Protégé-2000 mechanisms for modeling OIL ontologies as possible. If a new class is simply a subclass of several existing classes in the hierarchy, we use Protégé's own notion of subclasses by placing the new class where it belongs in the hierarchy. However, if a superclass definition requires boolean expressions—something Protégé-2000 does not allow—we use the subClassOf field that we see on the template. Even though Protégé-2000 does not understand the semantics of this field, we can represent this additional superclass information declaratively, and then pass it to a classifier or simply save in OIL.

We use the hasPropertyRestriction field when we need complex expressions or when we need to specify *existential* slot constraints: Protégé-2000 allows definition of value-type constraints on slots ("All values of this slot must be instances of this class"). OIL allows existential slot constraints in addition to the value-type constraints ("One value of this slot must come from this class and one value must come from that class"). We build the complex expressions declaratively by creating instances of core auxiliary classes. We can see some of these core classes in the tree in Figure 8. In the example, we specify a subclass of a meal-course, spicy-red-meat-course, which we define as "a course that must contain red meat *and* must contain food that is itself spicy *or* food containing something that is spicy."

Even though Protégé-2000 does not support some of the semantics that OIL has, we can still encode the additional information declaratively. Protégé-2000 will "ignore" the information, but it will be able to pass it on to a classifier or to encode it in OIL so that an OIL agent can understand it.

The user-interface level

Apart from changing labels and rearranging fields on the forms for the OilClass and OilProperty templates, we created a new slot widget to allow easier editing of nested expressions such as the ones representing "food that is itself spicy *or* food containing something that is spicy" in Figure 8. This widget augments the standard Protégé-2000 widget for selecting and creating values for instance-valued slots with a display of the nested expressions in a more practical form. A further extension of this simple but effective slot widget can include a full context-sensitive, validating expression editor.

The back-end level

We describe here an OIL back end that produces the RDFS output for OIL. Therefore, we can build largely on the existing RDFS back end. In defining the class and slot names and the structure of the auxiliary core classes in the OIL editor, we have mainly adhered to the RDFS specification of OIL. As a result, just using the RDFS back end, described earlier, gives us an output that is very close but not identical to the RDFS OIL output that we need. Thus, to create the OIL back end, we started with the existing RDFS back end. We adapted it to add the parts of definitions specified by the native Protégé-2000 means to the complex class expressions.

The OIL back end encodes the concepts that Protégé-2000 has and OIL does not (global cardinality restrictions on slots, for example) by defining additional statements in a Protégé namespace. An OIL agent will not understand these statements and will ignore them, but Protégé-2000 will be able to extract the necessary information from them.

Because many Semantic Web languages are in their infancy and already come in many different versions, there is an alternative approach to developing specific back ends for each of these versions. We can create a general RDF back end for Protégé-2000 and then use a declarative and easily adaptable RDF transformation language for generating the desired outputs. Some research groups are currently investigating such a back end and the corresponding RDF transformation (and query) language.

Accessing external services through a tab plug-in

The DL languages, such as OIL and DAML+OIL, traditionally rely on an inference component—a classifier—to find the

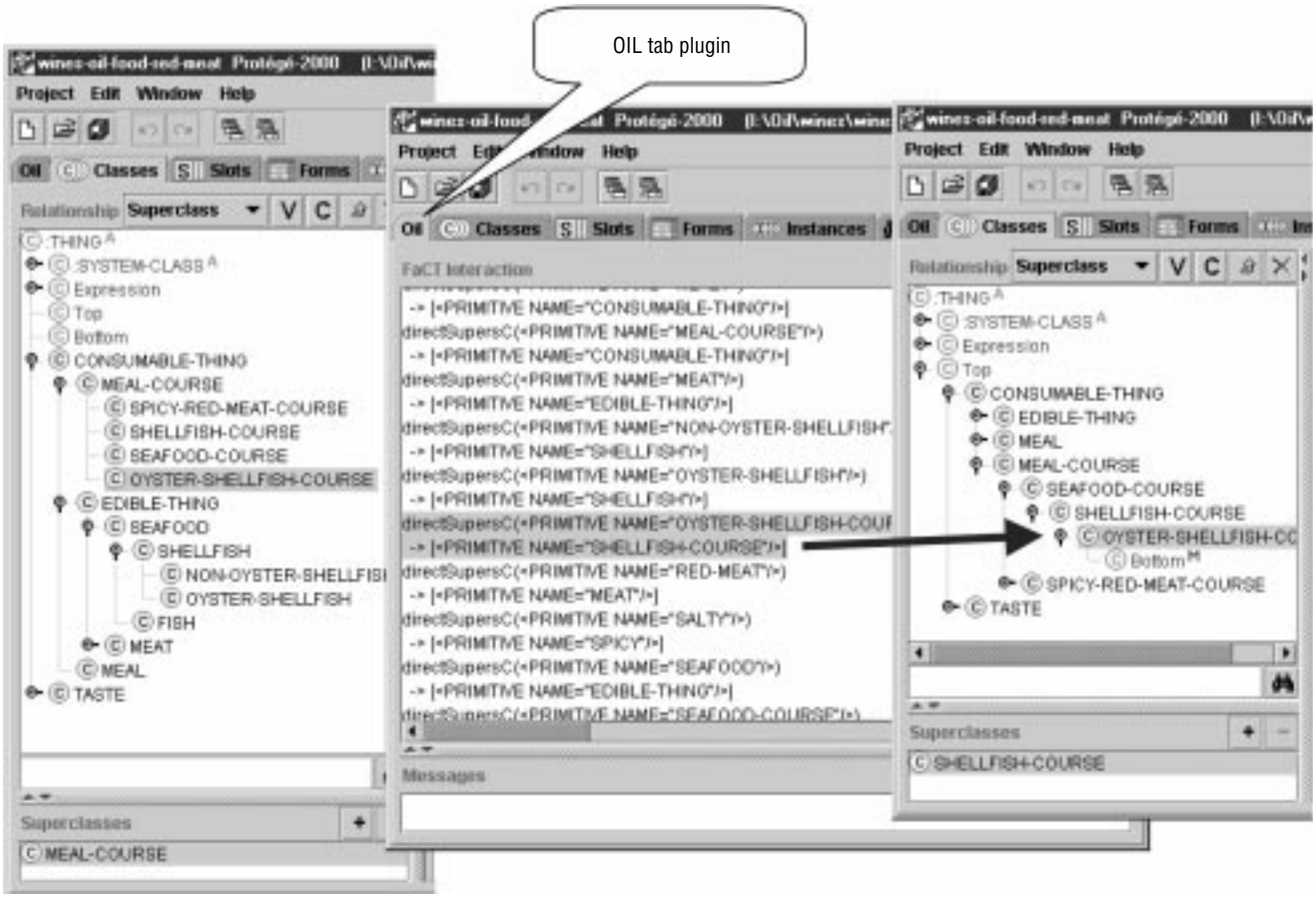


Figure 9. Tab plug-in for classification of OIL ontologies. On the right, we see a hierarchy of meal courses before classification. The middle pane shows interactions with the FaCT classifier. The hierarchy on the right is the one that the classifier computed.

right position of a class in the class hierarchy and to determine which class definitions are unsatisfiable (cannot have any instances). Therefore, it is crucial to have a connection to a DL classifier as part of the environment for editing OIL and DAML+OIL ontologies. Having created a set of definitions, we can invoke the classifier to determine how the evolving class hierarchy looks. We can see the effects that changes in class definitions will have on the evolving hierarchy. We can immediately check if logical expressions defining a class contradict one another making the class unsatisfiable.

Therefore, in order to create a full-fledged Protégé-based OIL editor, we need to connect Protégé-2000 to such an inference component and present the results to the user. We implemented this connection as a Protégé-2000 tab plug-in.

Figure 9 shows the OIL tab in action. Initially, the class hierarchy has the various **meal-course** subclasses as siblings. In addition, we specify that an **oyster-shellfish-course** is a **meal-course** that has OYSTERS as the value for its FOOD slot; a **shellfish-course** is a **meal-course** that has **shellfish** as its **food**, and so on. We then use the OIL tab to connect to a DL classifier, FaCT,[6] and to have it rearrange the class hierarchy according to the class definitions. In the resulting hierarchy, the **oyster-shellfish-course** class, for example, is correctly classified as being a subclass of the **shellfish-course** class.

With the advent of the Semantic Web, the current network of online resources is expanding from a set of static documents designed mainly for human consumption to a new Web of dynamic documents, services, and devices, which software agents will be able to understand. Developers will likely create many different representation languages to embrace the heterogeneous nature of these resources. Some languages will be used to describe specific domains of knowledge; others will model capabilities of services and devices. These languages will have different emphasis, scope, and expressive power.

Protégé-2000 provides full-fledged support for knowledge modeling and acquisition. Developers also can custom-tailor Protégé-2000 quickly and easily to be an editor for a new Semantic Web language. A Protégé-based editor enables modeling at a conceptual level that allows developers to think in terms of concepts and relations in the domain that they are modeling and not in terms of the syntax of the final output.

By adapting Protégé-2000 to edit a new Semantic Web language rather than creating a new editor from scratch or using a text editor to create ontologies in the new language, we obtain a graphical, conceptual-level ontology editor and knowledge-acquisition tool. We get a new editor to experiment with the new language without investing many resources into it. And we can use Protégé-2000 as an *interchange module* to translate most of the models in other Semantic-Web languages into our new language and vice versa. In our experience, it takes a few days

to adapt Protégé-2000 to a new Semantic-Web language—a lot less time than is required to create any sophisticated software from scratch. We were able to create these editors even for a language like OIL, which takes a knowledge-modeling approach that is different from the frame-based approach for which Protégé originally was designed. The extensible and flexible knowledge model and the open plug-in architecture of Protégé-2000 constitute the basis for developing a suite of conceptual-level editors for Semantic Web languages.

Acknowledgments

For more information about the Protégé project, please visit http://protege.stanford.edu. A grant from Spawar, a grant from FastTrack Systems, and the DARPA DAML program supported this work.

References

1. T. Berners-Lee, M. Fischetti, and M. Dertouzos, *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web by its Inventor*, Harper, San Francisco, 1999.
2. D. Brickley and R.V. Guha, "Resource Description Framework (RDF) Schema Specification," World Wide Web Consortium, Proposed Recommendation 1999, www.w3.org/TR/2000/CR-rdf-schema-20000327 (current 28 Mar. 2001).
3. J. Hendler and D.L. McGuinness, "The DARPA Agent Markup Language, " *IEEE Intelligent Systems*, vol. 16, no. 6, Jan./Feb., 2000, pp. 67–73.
4. N.F. Noy, R.W. Fergerson, and M.A. Musen, "The Knowledge Model of Protégé-2000: Combining Interoperability and Flexibility," *Proc. Knowledge Engineering and Knowledge Management: 12th Int'l Conf. (EKAW-2000), Lecture Notes in Artificial Intelligence*, no. 1937, Springer-Verlag, Berlin, 2000, pp.17–32.
5. M.A. Musen et al., "Component-Based Support for Building Knowledge-Acquisition Systems, " *Proc. Conf. Intelligent Information Processing (IIP 2000) Int'l Federation for Information Processing World Computer Congress (WCC 2000)*, Beijing, China, 2000, http://smi-web.stanford.edu/pubs/SMI_Abstracts/SMI-2000-0838.html (current 28 Mar. 2001).
6. I. Horrocks, "The FaCT system," *Proc. Automated Reasoning with Analytic Tableaux and Related Methods: Int'l Conf. Tableaux 98, Lecture Notes in Artificial Intelligence,* no. 1397, Springer-Verlag, Berlin, 1998, pp. 307–312.

The Authors

Natalya F. Noy is a research scientist in the Stanford Medical Informatics laboratory at Stanford University. Her interests include ontology development and evaluation, semantic integration of ontologies, and making ontology-development accessible to experts in noncomputer-science domains. She has a BS in applied mathematics from Moscow State University, Russia, an MA in computer science from Boston University, and a PhD in computer science from Northeastern University in Boston. Contact her at Stanford Medical Informatics, 251 Campus Dr., Stanford Univ., Stanford, CA 94305; noy@smi.stanford.edu.

Michael Sintek is a research scientist at the German Research Center for Artificial Intelligence. Currently, he is project leader of the FRODO project where an RDF-based framework for building distributed organizational memories is developed. He has a Diplom in computer science and economics from the University of Kaiserslautern. Contact him at the German Research Center for Artificial Intelligence (DFKI) GmbH, Knowledge Management Group, Postfach 2080, D-67608 Kaiserslautern, Germany; sintek@dfki.uni-kl.de.

Stefan Decker is a postdoctoral fellow at the Department of Computer Science at Stanford University, where he works on Semantic Web Infrastructure in the DARPA DAML program. His research interests include knowledge representation and database systems for the Web, information integration, and ontology articulation and merging. He has a PhD in computer science from the University of Karlsruhe, Germany, where he worked on ontology-based access to information. Contact him at Stanford University, Gates Hall 4A, Room 425, Stanford, CA 94305; stefan@db.stanford.edu.

Monica Crubézy is a postdoctoral researcher in the Stanford Medical Informatics laboratory at Stanford University. Her research focuses on the modeling and integration of libraries of problem-solving methods in the Protégé knowledge-based-system development framework. She graduated from the École Polytechnique Féminine, France, in general engineering and computer science. She has a PhD in computer science from the Institut National de Recherche en Informatique et Automatique in Sophia Antipolis, France. Contact her at Stanford Medical Informatics, 251 Campus Drive,Stanford University, Stanford, CA 94305; crubezy@smi.stanford.edu.

Ray Fergerson is a programmer in the Stanford Medical Informatics laboratory at Stanford University. He has a BS in physics from the Colorado School of Mines and a PhD in experimental nuclear physics from the University of Texas in Austin. Contact him at Stanford Medical Informatics, 251 Campus Drive, Stanford University, Stanford, CA 94305; fergerson@smi.stanford.edu.

Mark A. Musen's biography appears in the Guest Editors' Introduction on page 25.