The knowledge model of Protégé-2000: combining interoperability and flexibility

Natalya Fridman Noy, Ray W. Fergerson, Mark A. Musen
Stanford Medical Informatics, Stanford University, Stanford, CA 94305-5479
{noy, fergerson, musen}@smi.stanford.edu

Abstract

Knowledge-based systems have become ubiquitous in recent years. The World-Wide Web consortium is developing the Resource Description Framework (RDF)—a system for annotating even Web pages with knowledge elements. Knowledge-base developers need to be able to share and reuse knowledge bases that they build. Therefore, interoperability among different knowledge-representation systems is essential. The Open Knowledge-Base Connectivity protocol (OKBC) is a common query and construction interface for frame-based systems that facilitates this interoperability. Protégé-2000 is an OKBC-compatible knowledge-base–editing environment developed in our laboratory. Protégé-2000 has an easy-to-use and configurable interface. We describe its OKBC-compatible knowledge model that makes the import and export of knowledge bases from and to other knowledge-base servers easy. We discuss how the requirements of being usable and configurable knowledge-acquisition tool affected our decisions in the knowledge-model design. Protégé-2000 also has a flexible metaclass architecture which provides configurable templates for new classes in the knowledge base. The use of metaclasses makes Protégé-2000 easily extensible and enables its use with other knowledge models. For example, we demonstrate that we can resolve many of the differences between the knowledge models of Protégé-2000 and RDF by defining a new metaclass set. Resolving the differences between the knowledge models in declarative way enables easy adaptation of Protégé-2000 as an editor for other knowledge-representation systems.

1 The trade-off between interoperability and usability of knowledge-based systems

In recent years, knowledge sharing and reuse has become one of the primary goals of the knowledge-based systems research community [12]. Enabling interoperability among knowledge-representation systems is a crucial step in achieving this goal. The Open Knowledge-Base Connectivity (OKBC) protocol [2, 7] facilitates this interoperability by providing an application-programming interface (API) that serves as a common query and construction interface for frame-based systems. A number of OKBC-compatible knowledge-representation systems is currently available including Ontolingua [5], Loom [11], and Protégé-2000, which was developed in our laboratory.

Protégé-2000 is the latest component-based and platform-independent generation of the Protégé toolset [6]. Two goals have driven the design and development of Protégé-2000: (1) achieving interoperability with other knowledge-representation systems, and (2) being

an easy-to-use and configurable knowledge-acquisition tool. We achieve the first goal by making the knowledge model of Protégé-2000 compatible with OKBC (Section 2). As a result, Protégé-2000 users can import ontologies from other OKBC-compatible servers and export their ontologies to other OKBC knowledge servers. Protégé-2000 uses the freedom allowed by the OKBC specification to maintain the model of structured knowledge acquisition (Section 3) that was present in all the generations of the Protégé tools and to achieve the second design goal of being a usable and extensible tool.

To function effectively as an access layer for many different knowledge-representation systems, it was important for OKBC to have an extremely general knowledge model. The set of representational commitments in the OKBC knowledge model is minimal and OKBC allows knowledge-representation systems to define their own behavior for many aspects of the knowledge model (e.g., default slot values). Protégé-2000 *restricts* the OKBC knowledge model in the following two ways (Section 4):

(1) If implementing a general feature caused significant changes to the way that knowledge acquisition is performed in Protégé and if OKBC allowed restricting the feature and still remain OKBC-compatible, Protégé-2000 did that.
(2) If OKBC did not specify the behavior of a knowledge-model component at all, Protégé-2000 specified the behavior explicitly.

Protégé-2000 also *extends* some features of the OKBC knowledge model in a way that is not prohibited by OKBC. The Protégé-2000 metaclass architecture is one such extension (Section 5). The use of metaclasses allows Protégé-2000 to apply the knowledge-acquisition approach that it uses to acquire instance data to the ontology-editing process itself. One of the central user-interface metaphors in Protégé-2000 is the use of forms to acquire instance data. Protégé-2000 generates these forms automatically based on the class definitions and the users can then custom-tailor the forms if necessary. Protégé-2000 uses metaclasses widely and it implements the internal structure of its own knowledge model in a metaclass architecture. A **metaclass** is a template that is used to define new classes in an ontology. Therefore, we can customize and lay out the forms for specifying classes and slots in exactly the same way that we customize and lay out forms for acquiring instances. The metaclass architecture in Protégé-2000 along with its component-based approach also enables developers to use Protégé-2000 as an editor for knowledge-representation systems with knowledge models different from that of Protégé-2000.

We have demonstrated this flexibility by defining a knowledge model of Resource Description Framework (RDF) [13] as a new metaclass architecture in Protégé-2000. RDF, developed by the World-Wide Web consortium, is an emerging standard for defining metadata for encoding machine-readable semantics in Web documents. The knowledge model underlying RDF—RDF Schema [1]—is different from the Protégé-2000 knowledge model. However, it is possible to define the main elements of the RDF knowledge model by defining the metaclasses that will add RDF-specific features to the templates used to create new classes and slots (Section 6). This definition enables the creation and editing of RDF documents in Protégé-2000. Our collaborators have implemented an independent Protégé-2000 component that translates the RDF knowledge

base created in Protégé-2000 into standard RDF syntax effectively making Protégé-2000 an editor for RDF documents.

2 Protégé-2000 knowledge model

The knowledge model of Protégé-2000 is frame-based: frames are the principal building blocks of a knowledge base. A Protégé **ontology** consists of classes, slots, facets, and axioms. **Classes** are concepts in the domain of discourse. **Slots** describe properties or attributes of classes. **Facets** describe properties of slots. **Axioms** specify additional constraints. A Protégé-2000 **knowledge base** includes the ontology and individual **instances** of classes with specific values for slots. The distinction between classes and instances is not an absolute one, however, as we will discuss shortly.

We will use the task of modeling the knowledge about a newspaper-publishing company as an example throughout the paper.[1] This model must include such notions as employees and authors of a newspaper, the newspaper itself, different newspaper sections, layout of issues for each day of the week, and so on. We do not attempt to build a comprehensive model of this domain but rather we use it solely as an illustration of the concepts we discuss. In this domain, we can define a *class* `Newspaper`, where we will store the general information of what a newspaper is and what properties it may have. The January 1, 2000 issue of the *New York Times* will then be an individual *instance* of the class `Newspaper`.

2.1 Classes and instances

Classes in Protégé-2000 constitute a taxonomic hierarchy. If a class `A` is a subclass of a class `B` then every instance of `A` is also an instance of `B`. For example, a class representing newspaper `Editors` is a subclass of the `Employee` class (Figure 1). Protégé-2000 visualizes the subclass relation in a tree. Protégé supports multiple inheritance: one class can have more than one superclass. For example, `Editor` is subclass of both `Employee` and `Author`, since a newspaper editor is both an employee and an author of the newspaper. The root of the class hierarchy in Protégé-2000 (and in OKBC) is the built-in class `:THING`.

In Protégé-2000, both individuals and classes themselves can be **instances** of classes. A **metaclass** is a class whose instances are themselves classes (see Section 5).

2.2 Slots

Slots in Protégé describe properties of classes and instances, such as contents of a newspaper, or the name of an author. A slot itself is a frame. In Protégé, as in OKBC, slots are first-class objects: Slots are defined independently of any class. When a slot is

[1] You can browse the complete example of the newspaper ontology at the Protégé-2000 Web site: http://www.smi.stanford.edu/projects/protege/protege-2000/doc/users_guide/index.html

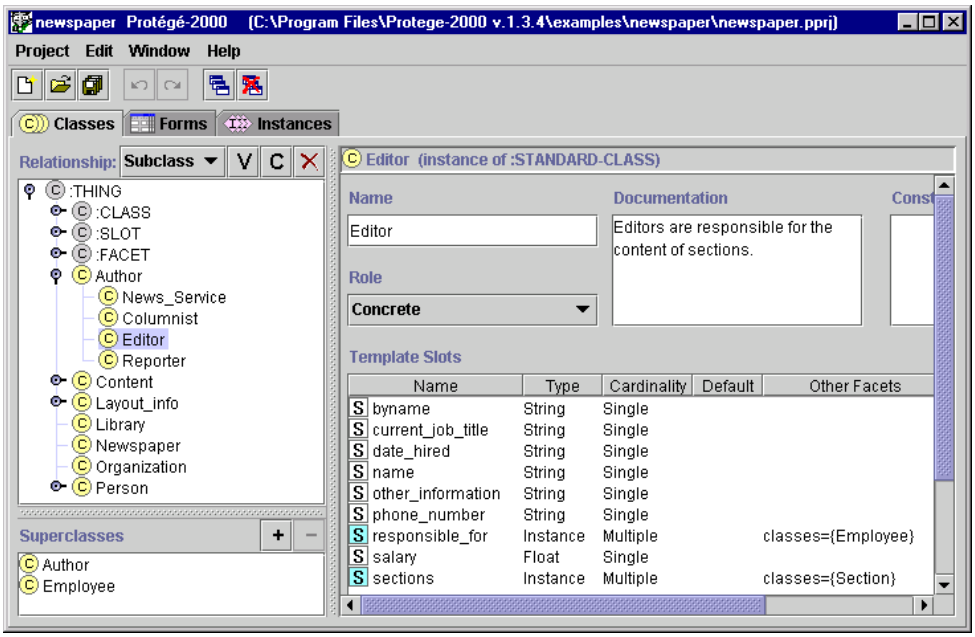

Figure 1. The representation of ontology in Protégé-2000. The left-hand panel contains the class hierarchy. The selected class `Editor` *is a subclass of two classes:* `Employee` *and* `Author`*. The right-hand panel is the form for the class* `Editor` *containing the own slots for the class and their values and the template slots attached to the class along with their value restrictions—facets.*

attached to a frame in the user's ontology, it describes properties of that particular frame. For example, we can define a slot `name` and attach it both to the class `Newspaper` and to the class `Author` to represent the name of a newspaper and the name of an author respectively. When a slot is attached to a frame it can have a **value**. For example, the `name` slot for a specific issue of a newspaper (instance of the `Newspaper` class) may have a string "New York Times" as the value.

2.3 Facets

One way to specify *constraints* on allowed slot values is through **facets**. The constraints specified using facets include cardinality of a slot (how many values the slot can have), restrictions on the value type of the slot (for example, integer, string, instance of a class), minimum and maximum value for a numeric slot, and so on. Facets define restrictions on an attachment of a slot to a class frame. For example, a slot `salary` can have type `Float` and minimum value of `15,000`. When the slot `salary` is attached to the class `Editor`, we can increase the minimum value to `30,000`.

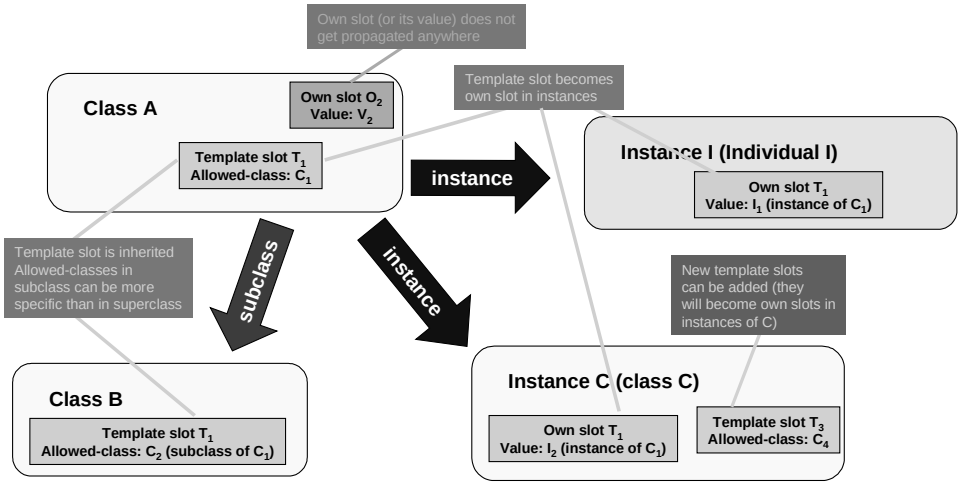

Figure 2. Propagation of template and own slots through the `subclass-of` *and* `instance-of` *relations.*

2.4 Template and own slots

A slot can be attached to a frame in one of two ways: as a **template slot** or as an **own slot**. An own slot attached to a frame describes properties of an object represented by that frame (an individual or a class). Own slots attached to a class do not get inherited to its subclasses or propagated to its instances. Template slots can be attached only to class frames. A template slot attached to a class is inherited by its subclasses. In addition, a template slot on a class becomes an own slot on the instances of that class (Figure 2).

For example, a slot containing a name of a specific editor—a `name` slot attached to a frame representing that individual instance of the class `Editor`—is an *own slot* attached to that frame (Figure 2). All the other slots on the instance frame—`salary`, `date hired`, and so on—are also own slots attached to this instance.

Classes can have own slots as well. For example, `documentation` for a class is an own slot attached to that class since it describes the class itself rather than instances of that class. If we represented synonyms for each class name, then the `synonyms` slot would also be an own slot for a class: "Source" is a synonym for the class name `Author` in the newspaper context, but it is not a synonym for John—a specific instance of the class `Author`. Since both `documentation` and `synonyms` in this example are own slots for a class, these slots and their values do not get inherited by subclasses. Indeed, synonyms of the class name `Author` are not related to the synonyms for class name `Columnist`—a subclass of `Author`.

Template slots describe properties that an instance of a class shall have. For example, instances of the class `Editor`—individual editors themselves—have names, dates when

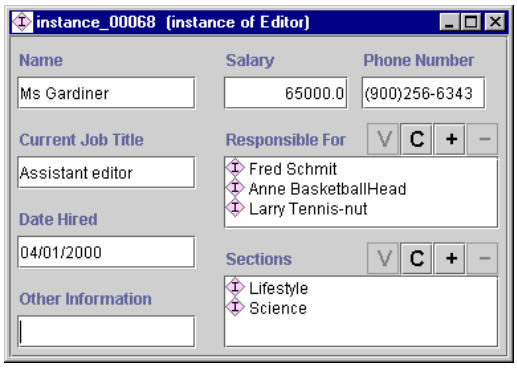

Figure 3. An instance of a class `Editor`*. All the slots on the form are own slots.*

they were hired, newspaper sections for which they are responsible, and so on. The slots `name`, `date hired`, and `sections` are template slots for the class `Editor` (Figure 1). Every instance of the class `Editor` has these slots as own slots with specific values (Figure 3). Any subclass of `Editor` also will inherit these template slots. In fact, the class `Editor` inherited most of its template slots from its superclasses: `Author` and `Employee` (see Figure 1).

To summarize, own slots describe a property of a (class or individual) frame itself rather than properties of instances of that frame. Template slots describe properties of instances of a class. Own slots do not propagate to either subclasses or instances of the frame to which they are attached. Template slots get inherited as template slots to the subclasses, and they become own slots for instances.

Protégé-2000 does not allow direct attachment of own slots to classes or instances (we explain why in Section 4). An individual instance can acquire own slots only by being an instance of a class that has those slots as template slots and a class can acquire own slots only by being an instance of a metaclass that has those slots as template slots.

3 Knowledge-acquisition forms

In Protégé-2000, s*tructured data entry* allows users to enter information about instances quickly and easily and to verify the entered information directly. Protégé-2000 enables structured data entry by using knowledge-acquisition **forms** to acquire instances information. The form-based interface is one of the central user-interface metaphors in Protégé. When a user defines a class and attaches template slots to it, Protégé automatically generates a form to acquire instances of that class. The slots for the class, their cardinality and value type determine the default layout and the content of the form. For example, Protégé-2000 uses a text field as a default way to display and acquire a value of a single-cardinality slot of type `string`. It uses a pull-down menu for a slot whose values are limited to a set of symbols. It represents a boolean slot with a checkbox. It uses a list for slots that have multiple values.

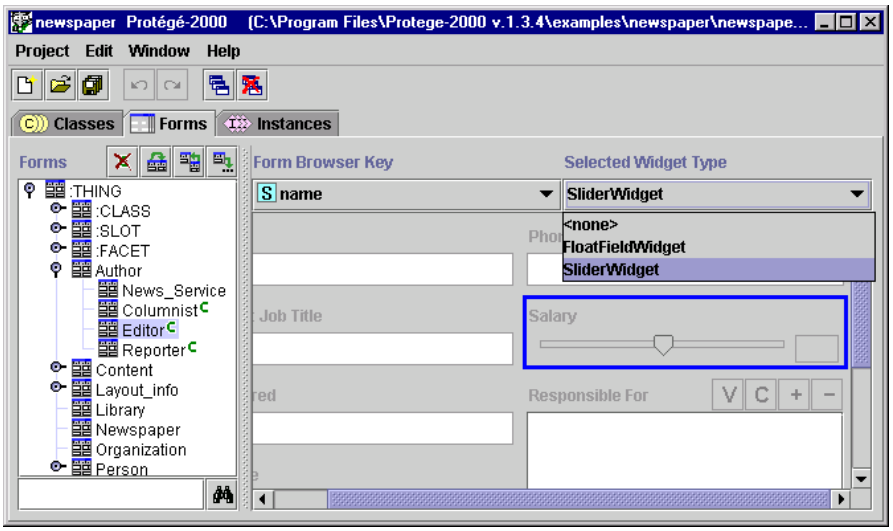

Figure 4. Customizing the form for class `Editor`*. We choose to use a slider to acquire the numeric value for the* `salary` *slot instead of a text field used in the form in Figure 1.*

Users can customize the standard form that Protégé-2000 automatically generates for each class to suit the requirements of the specific class better. Customization includes changing the layout of the form components by moving the "important" information to the top of the form, changing labels for the form fields, and choosing different ways of displaying and acquiring slot values. For example, we can use a text field (with the appropriate validation) to acquire an integer value or we can use a slider to set the value instead (Figure 4). The component-based architecture of Protégé-2000 enables developers to write their own domain-specific and task-specific components to acquire and display slot values.

The current knowledge-acquisition process in Protégé-2000 consists of three steps: (1) define a class and its template slots; (2) lay out the form that will be used to acquire instances of that class; (3) acquire instance of the classes. Therefore, there is a form associated with each class and this form is used to acquire instances of that class. This user-interface approach resulted in some restrictions in our support of the OKBC knowledge model. We discuss these restrictions in the next section.

4 OKBC and Protégé knowledge models

Protégé-2000 uses the OKBC knowledge model [3] as the basis for its own knowledge model. The main goal of the OKBC developers was to ensure maximum generality and interoperability among knowledge-representation systems. Therefore the OKBC knowledge model makes minimal knowledge-representation commitments and it is extremely general. OKBC incorporates all the features of the basic approaches to frame-

based systems. Some of these features contradict each other or are not mutually compatible. The OKBC goal is to allow maximum flexibility. Therefore, when a conflict or incompatibility between two features arose, the OKBC designers often *allowed* both features without *requiring* the OKBC-compatible systems to implement either of them.[2] In some cases, OKBC did not specify the behavior at all leaving the implementation up to the knowledge-representation systems completely. This approach—allowing as many features as possible, requiring as few features as possible, and leaving some features underspecified—is perhaps the best approach for a common-access protocol, but designers of individual systems must make some design choices that restrict this generality.

The Protégé-2000 knowledge model is completely compatible with OKBC: Protégé implements everything that the OKBC knowledge model requires and everything in the Protégé-2000 knowledge model is logically consistent with OKBC. For the features where OKBC allows flexibility, Protégé-2000 preserves as much of the OKBC generality as possible and restricts the OKBC model only when the current Protégé user-interface paradigm requires it. Table 1 summarizes where the Protégé-2000 knowledge model is less general than the OKBC model. The list in Table 1 demonstrates the tradeoffs between an extremely general knowledge model (OKBC) and a system designed for knowledge acquisition (Protégé-2000): we had to sacrifice some of the generality of OKBC to maintain easy-to-use and configurable knowledge-acquisition interface. Table 1 does not include all the differences between the knowledge models of Protégé-2000 and OKBC. The exhaustive discussion of features where Protégé-2000 knowledge model diverges from the OKBC one, such as own facets and template-slot values, is beyond the scope of this paper.

	OKBC	**Protégé-2000**
1	A frame can be an *instance of multiple classes*	A frame can be an *instance of only one class*
2	A frame *does not have to be an instance* of any class	A frame is *always an instance* of a class
3	An *own slot can be attached directly* to any frame	An *own slot* attached to a frame is always *derived from the corresponding template slot*
4	Classes, slots, facets, and individuals *do not have to be frames*	*Every class, slot, facet, and individual is a frame*
5	A frame can be *a class, a slot, and a facet at the same time*	A frame is *either a class, or a slot, or a facet*

Table 1. Summary of differences between the knowledge models of OKBC and Protégé-2000.

[2] The OKBC protocol enables the knowledge-based systems to specify the design choices that they make.

We made the design choices in Protégé-2000 to enable optimal knowledge acquisitions. As with any evolving system, the decisions may change if we no longer believe that they are justified by the knowledge-acquisition requirements.

The first three differences in Table 1 result from the Protégé approach to modeling and knowledge acquisition: Protégé generates a knowledge-acquisition interface based on the users' specification of the objects that they need to acquire. This specification includes defining a class of objects to acquire, having Protégé generate a knowledge-acquisition form based on the class definition, and, if necessary, custom-tailoring the form to acquire that class of objects (Section 3). This approach to knowledge acquisition requires that there is always a way to specify and to custom-tailor a form for acquiring information for any instance. The first four items in Table 1 ensure that there is indeed a class form associated with every instance. If an instance has more than one direct type, there is no single place where the form can be laid out, since different own slots for the instance come from different classes. Similarly, if an own slot is attached directly to an instance and it does not come from a template slot, there is no place where the user can customize how the values of the slot should be acquired. The same problem arises if a frame is not an instance of *any* class: all the own slots need to be attached to it directly and there is no place where the form can be laid out before the slot values for the instance are acquired.

Currently, the Protégé-2000 user interface is centered around frames and forms to acquire values for slots. Therefore only primitive data types—numbers, strings, symbols, and so on—do not have to be represented using frames. All the user-defined entities are frames. This requirement is less general than the OKBC approach where objects do not have to be frames. For example, a set {1, 2, 3, 4} can be a class.

The sets of classes, slots, and facets are disjoint in Protégé-2000. An object can be either a class, or a slot, or a facet. Protégé imposes this requirement because the three types of frames play inherently different roles in the knowledge acquisition process. Classes are collections of terms organized in the hierarchy and each class defines what knowledge will need to be acquired for its instances.

In summary, we used the flexibility that OKBC allows to restrict some of the features of the extremely general knowledge model and thereby maintain the uniform knowledge-acquisition interface. There are aspects of a knowledge-representation system for which OKBC allows alternative behaviors or does not specify the behavior at all (leaving it up to the individual systems to implement any behavior and still to remain compatible with the OKBC). We utilized this flexibility to implement the Protégé metaclass architecture described in the next section.

5 Metaclasses

The Protégé-2000 metaclass architecture enables us to extend the knowledge-acquisition approach that we use to acquire instance data to the ontology-editing process itself. We can customize and lay out the forms for specifying classes and slots in exactly the same way that we customize and lay out forms for acquiring instances (Section 3). The metaclass architecture in Protégé-2000 also enables developers to use Protégé-2000 as an

editor for knowledge-representation systems with knowledge models different from that of Protégé-2000. For example, in Section 6 we describe an example implementation of an editor for Resource Description Framework schema and instance data in Protégé-2000. The Protégé-2000 metaclass architecture is modeled after the metaobject protocol [9] in the Common Lisp Object System [8].

A **metaclass** is a class whose instances are themselves classes. Every frame in Protégé-2000 is an instance of a class (see Table 1). Since classes are also frames, every Protégé-2000 class is an instance of another class. Therefore, every class has a dual identity: It is a *subclass* of a class in the class hierarchy—its superclass,—and it is an *instance* of another class—its metaclass. In this section, we will describe how metaclasses work in Protégé-2000, give examples of how metaclasses can be used in knowledge modeling and describe the metaclass architecture of the Protégé-2000 knowledge model itself.

5.1 What is a metaclass

Metaclass is a *template* for classes that are its instances. A metaclass describes how a class that instantiates this template will look: namely, which own slots it will have and what are the constraints for the values of these slots. Similarly, a traditional class describes how instances of that class will look: which own slots the instances will have and what are the constraints for the values of these slots.

Own slots for a class—the slots that the class acquires from its metaclass—describe the properties of the class itself and not of its instances. For example, a class's `role`—concrete or abstract—defines whether or not the class can have direct instances. (Abstract classes do no have direct instances.) This property is an intrinsic property of the class itself and not the property of its instances. Similarly, the value of the `documentation` slot for a class describes the class itself and not the class' subclasses or instances.

In Protégé (and in OKBC) all metaclasses inherit from the system class `:CLASS`. By default, each class in Protégé is an instance of the `:STANDARD-CLASS` metaclass, which is a subclass of `:CLASS`. The `:STANDARD-CLASS` metaclass has template slots to store a class's name, documentation, a list of template slots, a list of constraints, and so on (Figure 7). These slots then become own slots for each of the newly created classes—instances of `:STANDARD-CLASS`. We will discuss `:STANDARD-CLASS` in more detail in Section 5.3. The form that Protégé uses to acquire the class information (for example, the form on the right-hand side in Figure 1) is in fact the knowledge-acquisition form that corresponds to the `:STANDARD-CLASS` class. Users can customize this form in exactly the same manner as they customize forms for other classes.

Protégé-2000 allows users to define their own metaclasses and to define new classes as instances of these user-defined metaclasses. They can then customize the forms to acquire instances of these metaclasses, which are new classes in the ontology, effectively creating new ontology editors [4].

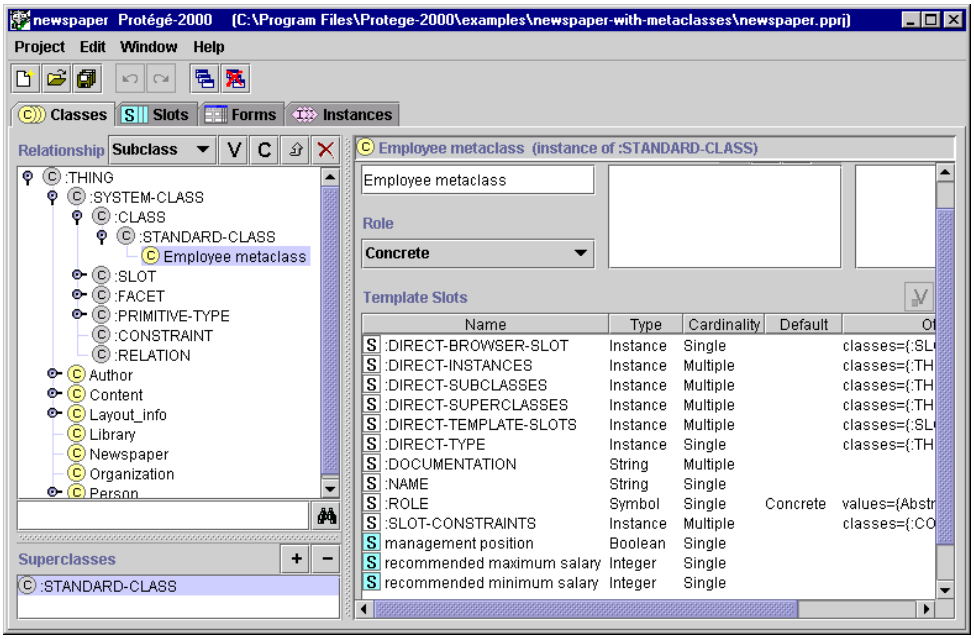

Figure 5. Definition of the class `Employee metaclass`*. The template slots defined here will become own slots for classes that are instances of this metaclass.*

5.2 User-defined metaclasses

Consider the earlier example of defining a newspaper ontology. Suppose we wanted to record the recommended minimum and maximum salary for different employee positions as well as whether or not a particular position is at the level of management. For example, the recommended minimum salary for an editor is $30,000 and the recommended maximum salary is $60,000. An editor is a management position. Since the two numbers are only the *recommended* minimum and maximum, we cannot encode them as explicit minimum and maximum constraints for the `salary` slot: our model should allow a very good editor to earn more than $60,000. Also, we consider all editors to be managers and we should not have to specify this fact for each new instance of the `Editor` class. These three slots—the recommended minimum and maximum salary and whether or not the position is a management position—are in fact own slots for the class `Editor`: the values of these slots define properties of the `Editor` class, but not the properties of their corresponding instances.

We will use metaclasses to implement this model. We define an `Employee metaclass` class which we will then use as a template for the class `Employee` and all of its subclasses. Protégé-2000 requires that all metaclasses be subclasses of the system class `:CLASS`. In practice, most metaclasses are subclasses of `:STANDARD-CLASS`,

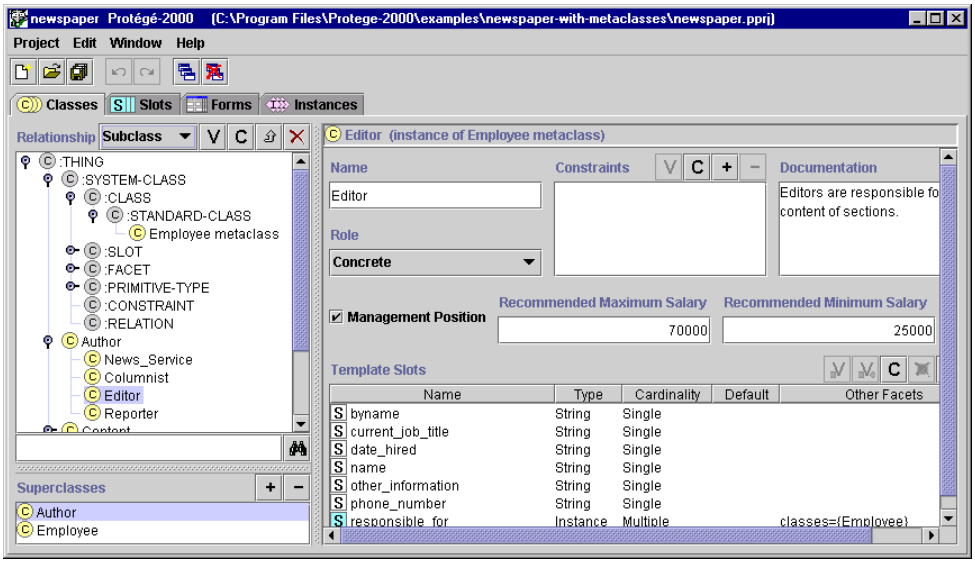

Figure 6. The definition of the `Editor` class as an instance of the `Employee` metaclass. The definition has the additional own slots with the corresponding values. The `Editor` class in Figure 1 was an instance of `:STANDARD-CLASS` and did not have these own slots.

since all the metaclasses usually include the slots defined in `:STANDARD-CLASS` (see Section 5.1). We create additional template slots for the `Employee metaclass` (Figure 5): the `minimum recommended salary` and `maximum recommended salary` slots have value type `Integer` and cardinality `Single`; the `management position` slot has the value type `Boolean`.

We define the `Employee` class and its subclasses as instances of the `Employee metaclass`. These classes then have the three new slots as their own slots. The `Editor` class in Figure 1 was an instance of the `:STANDARD-CLASS` metaclass and did not have these domain-specific own slots. Figure 6 shows the definition of the class `Editor` as an instance of `Employee metaclass`. Before defining the `Editor` class, we laid out the form for the `Employee metaclass` class in a manner similar to the one demonstrated in Figure 4. Protégé-2000 uses this form to acquire instances of the `Employee metaclass` class such as the `Editor` class.

5.3 Protégé-2000 built-in metaclass architecture

Protégé-2000 uses the metaclass mechanism to implement its own internal class structure. Metaclasses define the representation of all the frames in the system—classes, slots, facets, and individuals. All the information about the frames, from name and

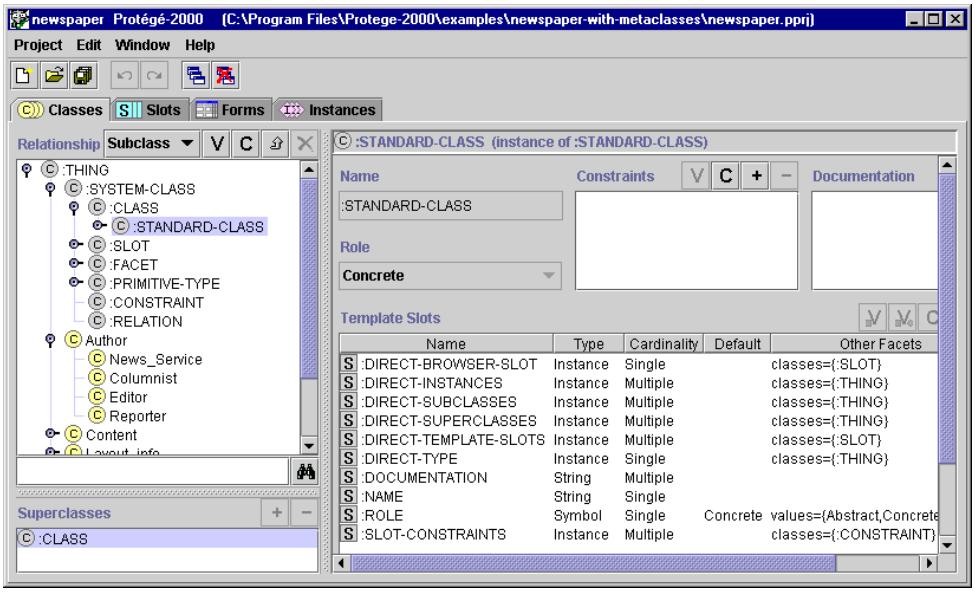

Figure 7. Definition of the `:STANDARD-CLASS` *metaclass. Protégé-2000 uses the slots attached to this class to store the information about all its classes.*

documentation of a class to a list of its template slots and superclasses, is stored in the class's own slots. In other words, Protégé uses its own class structure to store the information about itself. The users of Protégé-2000 do not need to see this internal information (and very few users do indeed see it) unless they decide to explore the ontology describing the Protégé-2000 knowledge model. The three system classes—`:CLASS`, `:SLOT`, `:FACET`—serve as types for all the Protégé-2000 classes, slots, and facets respectively. These classes do not have any template slots attached to them. This feature allows Protégé-2000 developers to implement their own knowledge models in Protégé-2000 without making the same knowledge-model assumptions that Protégé-2000 does if they do not need it. The Protégé-2000 knowledge model itself is implemented using the three standard subclasses of these classes: `:STANDARD-CLASS`, `:STANDARD-SLOT`, and `:STANDARD-FACET`. These three classes have the template slots that define the structure of their instances—the Protégé-2000 classes, slots, and facets respectively.

- `:STANDARD-CLASS` defines the default metaclass for Protégé-2000 classes. Template slots of `:STANDARD-CLASS` (Figure 7) define the standard own slots for classes in Protégé-2000. The slots store the class name, documentation, role, direct subclasses, direct superclasses, and direct template slots for the class. Usually, all the user-defined metaclasses will be subclasses of `:STANDARD-CLASS`. The `:STANDARD-CLASS` is an instance of itself and therefore has the

same sets of slots attached to it twice: once as own slots with values and once as template slots with value-type restrictions in the form of facets.

- `:STANDARD-SLOT` defines the default metaslot in Protégé-2000. Template slots of `:STANDARD-SLOT` (Figure 8) define the standard own slots for slot frames in Protégé-2000. These slots store the slot name, its documentation, value type, numeric minimum and maximum, cardinality, and constraints. User-defined metaslots are subclasses of `:STANDARD-SLOT`.
- `:STANDARD-FACET` define the class to which all the built-in and user-defined facets belong. Currently there are no slots required for facets.

Since all the metaclasses are at the same time regular classes in Protégé-2000, we can use the Protégé-2000 form mechanism to customize the forms for acquiring data for instances of these classes. For example, if a customized version of the system does not need to have the constraint or role slots on the class form, all that is necessary is to customize the form for `:STANDARD-CLASS` to remove the elements acquiring these values from the form.

To summarize, the metaclass architecture in Protégé-2000 enables users to adapt and change the knowledge model of the system to suit the requirements of their domain and task. This adaptation can be performed in two ways: (1) custom-tailoring the knowledge-acquisition forms for class definitions and (2) extending the Protégé-2000 knowledge model or defining a new knowledge model by extending the Protégé-2000 metaclass architecture or defining a new metaclass architecture respectively.

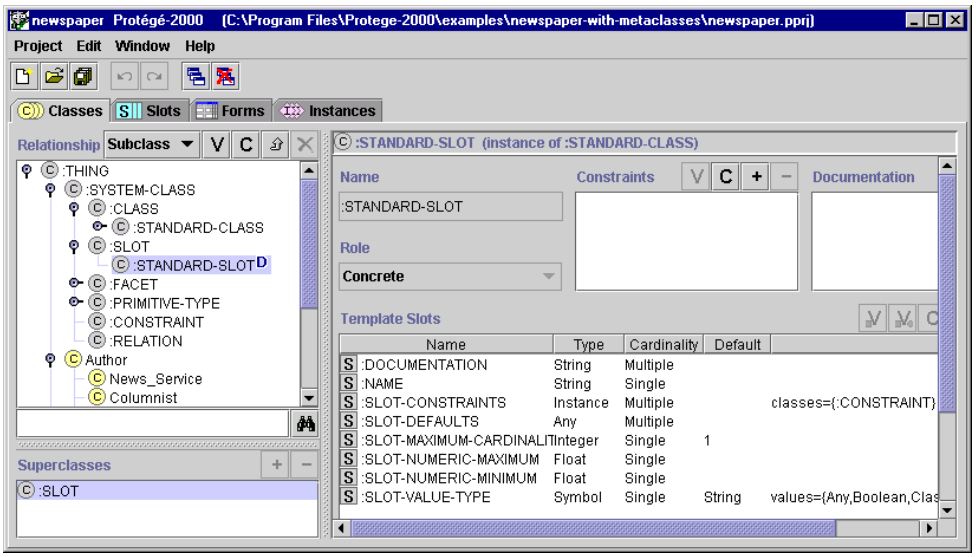

Figure 8. Definition of the `:STANDARD-SLOT` metaslot. Protégé-2000 uses the slots attached to this class to store the information about all its slots.

In the next section we demonstrate how we can use the Protégé-2000 metaclass architecture to implement a different knowledge model in Protégé-2000—that of RDF.

6 Adapting to a different knowledge model—RDF

The combination of (1) the metaclass architecture, (2) the ability to define specialized user-interface components to display and acquire slot values, and (3) the ability to implement additional persistence layers as components in Protégé-2000 allows Protégé-2000 users to adapt the tool to create and edit knowledge bases with knowledge models that are different from the Protégé-2000 knowledge model.

Together with our collaborators, we have recently adapted Protégé-2000 to become an editor for Resource Description Framework (RDF) Schema and RDF instance data. RDF is a knowledge representation standard being defined by the World-Wide Web Consortium [13]. The main goal for RDF is to make Web-based pages not only *human-readable* but also *machine-readable* enabling computer agents to use the information. Currently, HTML enables the Web-site developers to encode how the document is going to appear on the page. RDF takes the representation further, allowing semantics to be encoded in the markup tags as well. For example, two Web sites may describe the same movie: the Internet Movie Database has a list of all the characters, awards, technical details, and so on. Amazon.com has the information on purchasing the videotape. If the two sites represent the information formally and use a shared ontology to do so, an agent can link the two databases and can answer requests that require the knowledge of both. The RDF and RDF Schema specification define the knowledge model that will be used to express ontologies in RDF (RDF schema [1]) and the syntax to encode and instantiated the schema (RDF syntax [10]).

Adapting Protégé-2000 to become an editor for RDF Schema and instance data requires bridging the gap between the two knowledge models. We now describe what the differences are and how we can resolve many of the differences by defining specialized classes and metaclasses in Protégé-2000. We then discuss which differences we cannot resolve declaratively and describe how they can be solved as part of the RDF persistence layer—a Protégé-2000 component that translates between the Protégé-2000 internal representation of classes and the RDF serialization format.

6.1 Summary of the RDF knowledge model

RDF is a model for describing **resources**. Anything can be a resource. Each resource has a Uniform Resource Identifier (URI). An RDF document describes resources and the relations among resources. An RDF Schema defines what classes of resources exist, how resources are related to one another and what restrictions on those relations exist.

In RDF, **classes** of concepts constitute a hierarchy with multiple inheritance. Classes have instances and a resource can be an instance of more than class. Resources have **properties** associated with them. Properties describe attributes of a resource or a relation of a resource to another resource. A property is represented as a predicate–subject–object

triple: **predicate** is the property class, **object** and **subject** are resources related by the property. For example, a statement "John is a father of Alice" can be represent as a triple consisting of the predicate `father`, subject `John`, and object `Alice`. A property can be a specialization of another property. The RDF schema defines **domain** of a property—resources that can be subjects of the property—and **range** of a property—resources that can be objects of the property. For example, the property `father` may have a class `Person` both as its domain and as its range.

RDF resources have a set of core properties, which includes, for example the `rdfs:seeAlso` property, specifying another resource containing information about the subject resource and the `rdfs:isDefinedBy` property, indicating the resource defining the subject resource.

Therefore, a class (in the sense of frame-based languages) is defined by a resource and all the properties describing the resource (all the properties where the resource is the subject).

The following are the core classes in RDF:

- `rdfs:Resource`: the class containing all the resources—the superclass for all the RDF classes
- `rdfs:Class`: the class containing all classes as its instances—the metaclass for all the RDF classes
- `rdf:Property`: the class containing all the properties—the metaslot for the RDF properties

6.2 Core RDF classes and metaclasses in Protégé-2000

Both RDF and Protégé-2000 are based on a frame-based paradigm and their knowledge models have much in common. The notion of classes is the same. Classes are organized in a taxonomic hierarchy in both cases. Classes can have instances. *Properties* in RDF are *slots* in Protégé. Both properties and slots are first-class objects describing attributes of classes and instances and relations among them.

We start by defining in Protégé-2000 the core RDF classes described in the Section 6.1 (Figure 9). The Protégé-2000 class `rdfs:Resource` is the root of the hierarchy of classes defined in an RDF schema. Every class in the RDF project should be a subclass of `rdfs:Resource`. This class has three template slots: `resource` `URI` to store a URI of a resource and the core properties `rdfs:seeAlso` and `rdfs:isDefinedBy`, which are required by each resource. All the classes that are subclasses of `rdfs:Resource` inherit these slots from their parents.

The class `rdfs:Class` is a default metaclass for all the newly defined classes in the RDF project. It is also a subclass of `rdfs:Resource`. Therefore, `rdfs:Class` inherits the three template slots attached to `rdfs:Resource`. These template slots become own slots for instances of `rdfs:Class`—all the classes in the RDF project. The `rdfs:Resource` and `rdfs:Class` are themselves instances of `rdfs:Class` and therefore they have these standard slots as their own slots as well.

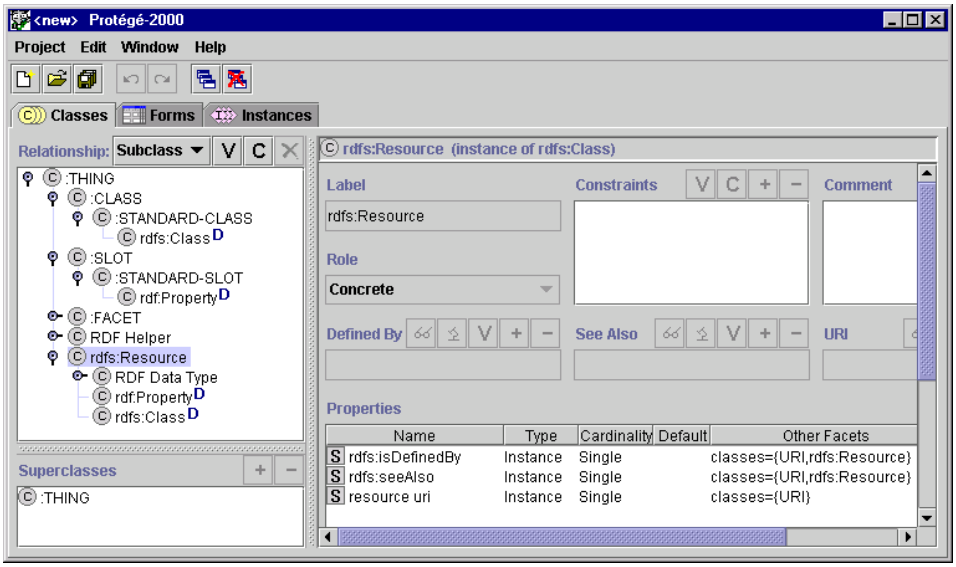

Figure 9. The RDF project: definition of the `rdfs:Resource` class. The template slots attached to this class will propagate to all the classes and instances in the project. The icon `D` at the `rdf:Property` class signifies that `rdf:Property` is the default metaslot.

The class `rdf:Property` is a default metaslot in the RDF project. It is the template for all newly defined slots. The metaslot `rdf:Property` is a subclass of both `rdfs:Resource` and `:STANDARD-SLOT`. As a result all slots—properties—have the standard own slots: `resource URI`, `rdfs:seeAlso`, and `rdfs:isDefinedBy`.

6.3 Differences between the Protégé-2000 and RDF knowledge models

Defining the RDF metaclass structure cannot resolve all the differences between the Protégé-2000 and RDF knowledge models. The implementation of the RDF persistence layer—a Protégé-2000 component that translates the knowledge base between the internal Protégé-2000 representation and RDF documents—can resolve the remaining differences between the knowledge models that could not be resolved by defining the metaclass structure. The RDF persistence layer enables the direct import and export of RDF documents making Protégé-2000 a graphical editor for RDF documents. Our collaborators have implemented one such component.

Table 2 summarizes the main differences between the Protégé-2000 and the RDF knowledge models that the persistence layer needs to resolve. As in OKBC, resources in RDF can have more than one type (that is an instance can belong to more than one class). In Protégé-2000 each instance is a member of only one class. However, *multiple inheritance* enables Protégé-2000 to simulate the multi-class membership. If an imported RDF schema has an instance `I` that belongs to several classes, C_1, …, C_n, the persistence

	Protégé-2000	RDF and RDF Schema
1	A frame can be an *instance of only one class*	A resource can be an *instance of multiple classes*
2	A value of a slot can be a value of a primitive type or an instance of a class. There can be *one or more* classes that constrain the value	The *range* of a property is a *single class* which constraints the objects of the property to instances of that class
3	*A slot* in an individual instance and *cannot be a subclass* of another slot	A property *can be a specialization of another property*
4	*Collections* (containers) are implemented as lists.	There are *three types of container objects*: bag, sequence, and alternative

Table 2. Summary of differences between the knowledge models of Protégé-2000 and RDF.

layer can create a new class `C` that is a subclass of all of C_1, …, C_n. All the template slots from C_1, …, C_n become template slots for the class `C`. The instance I will then be an instance of the class `C`—it will get all the template slots from all the original classes C_1, …, C_n as its own slots.

In RDF Schema, the range of a property is constrained to a single class. That is, the objects of a property have to be instances of that single class. In Protégé-2000, on the other hand, slot range can be a primitive data type, can be constrained to instances of several classes ("allowed classes" for the slot), or can be constrained to subclasses of several classes ("allowed parents" for the slot). The semantics of the range property in RDF Schema are modeled in Protégé-2000 by defining an `Instance` value type for the slot and listing only one class in the "Allowed Classes" facet. If a Protégé-2000 slot has multiple allowed classes, the RDF persistence layer can create a new class that is a superclass of all of the intended allowed classes and use the new class as the range for the RDF property.

The RDF Schema allows specializing properties. For example, the `father` property is a specialization of the more general `parent` property: If John is a `father` of Alice, then John is also a `parent` of Alice. In Protégé-2000, all slots are individual instances and therefore cannot form a subclass–superclass hierarchy. If a domain model requires a hierarchy of properties, the properties can be reified to become regular classes in the class hierarchy.

RDF Schema defines three different possible types of containers for properties that have multiple values: **sequence**, **alternative**, and **bag**. Protégé-2000 has a single (implicit) container type: that of a list. The RDF persistence layer can store all multiple-valued slots as RDF bags. The difference can be alleviated by implementing specialized knowledge-acquisition components that will be used to acquire and display containers of different types. The components will generate the appropriate internal representation for the various types of containers.

To summarize, we extended the Protégé-2000 built-in metaclass architecture to implement the structure of RDF Schema declaratively and the RDF persistence layer can use the knowledge structures to hide or repair other differences programmatically.

7 Conclusions

We have presented the knowledge model of Protégé-2000—an ontology-editing and knowledge-acquisition environment. The knowledge model's OKBC-compatibility enables the interoperability between Protégé-2000 and other OKBC-compatible systems. We have discussed how the knowledge model was affected by the requirements for structured knowledge acquisition. The Protégé-2000 metaclass architecture enables elegant and powerful knowledge modeling as well as allows us to implement the internal structure of Protégé-2000 explicitly in the ontology. We have described how Protégé-2000 uses its own metaclasses to describe itself. The flexibility of the knowledge model and the component architecture of Protégé-2000 make it easy to adapt the tool to work as an editor for other knowledge-representation systems. We demonstrated how Protégé-2000 can be adapted to become an editor for RDF.

Acknowledgments

Henrik Eriksson both inspired and was instrumental in the design of the Protégé-2000 metaclass architecture. Stefan Decker and William Grosso implemented the RDF persistence layer. We would also like to thank William Grosso for comments on the paper. This work was supported in part by the grants 5T16 LM0733 and 892154 from the National Library of Medicine, by a grant from Spawar, and by a grant from FastTrack Systems, Inc.

References

1. Brickley, D. and Guha, R.V., *Resource Description Framework (RDF) Schema Specification*, World Wide Web Consortium, 1999.
2. Chaudhri, V.K., Farquhar, A., Fikes, R., Karp, P.D. and Rice, J.P. OKBC: A programmatic foundation for knowledge base interoperability. In: *Fifteenth National Conference on Artificial Intelligence (AAAI-98)*. Madison, Wisconsin: AAAI Press/The MIT Press, 1998.
3. Chaudhri, V.K., Farquhar, A., Fikes, R., Karp, P.D. and Rice, J.P., *Open Knowledge Base Connectivity 2.0.3*., 1998.
4. Eriksson, H., Fergerson, R.W., Shahar, Y. and Musen, M.A. Automatic generation of ontology editors. In: *Twelfth Banff Workshop on Knowledge Acquisition, Modeling, and Management*. Banff, Alberta, 1999.
5. Farquhar, A., Fikes, R. and Rice, J., The Ontolingua server: a tool for collaborative ontology construction. *International Journal of Human-Computer Studies*. **46**: p. 707-727, 1997.

6. Grosso, W.E., Eriksson, H., Fergerson, R.W., Gennari, J.H., Tu, S.W. and Musen, M.A. Knowledge modeling at the millennium (the design and evolution of Protégé-2000). In: *Twelfth Banff Workshop on Knowledge Acquisition, Modeling, and Management*. Banff, Alberta, 1999.
7. Karp, P.D., Chaudhri, V.K. and Paley, S.M., A Collaborative Environment for Authoring Large Knowledge Bases. *Journal of Intelligent Information Systems*(*in press*), 2000.
8. Keene, S.E. and Gerson, D., *Object-Oriented Programming in Common LISP: A Programmer's Guide to CLOS*. Reading, Mass.: Addison-Wesley. xix, 266, 1989.
9. Kiczales, G., des Rivieres, J. and Bobrow, D.G., *The Art of the Metaobject Protocol*. Cambridge, MA: The MIT Press, 1991.
10. Lassila, O. and Swick, R.R., *Resource Description Framework (RDF) Model and Syntax Specification*, World Wide Web Consortium, 1999.
11. MacGregor, R., *Retrospective on Loom*, USC ISI, 1999.
12. Musen, M.A., Dimensions of knowledge sharing and reuse. *Computers and Biomedical Research*. **25**: p. 435-467, 1992.
13. W3C, *Resource Description Framework (RDF)*, World-Wide Web Consortium, http://www.w3.org/RDF/, 2000.