Automatic Generation of Ontology Editors

Henrik Eriksson

Department of Computer and Information Science
Linköping University
SE-581 83 Linköping, Sweden

E-mail: her@ida.liu.se

Raymond W. Fergerson Yuval Shahar Mark A. Musen

Stanford Medical Informatics
Stanford University
Stanford, California 94305-5479, U.S.A.

Abstract

Metalevel tools can support the knowledge-engineering process by assisting developers in the design and implementation of domain-oriented knowledge-acquisition tools. The use of ontologies as a basis for automatic generation of knowledge-acquisition tools simplifies the tool-specification process by taking advantage of ontologies defined as part of the knowledge-engineering process.

One of the drawbacks of this approach is that it separates ontology definition (in ontology editors) from instance editing (in knowledge-acquisition tools). Because many application tasks require ontology definition by domain experts, we have experimented with extending the Protégé framework to generate ontology editors in addition to knowledge-acquisition tools for instances. We have explored different approaches to ontology-editor specification in a series of prototype extensions to Protégé. Here, metaclass and metaslot definitions are the basis for ontology editors, which can be embedded in knowledge-acquisition tools and can be used by domain experts.

1 Introduction

The use of *metatools* to generate domain-specific knowledge-acquisition tools has proved to be a successful approach to providing appropriate tool support with a minimal tool-development effort. Several metatools assist the developer in the creation of custom-tailored knowledge-acquisition tools (Eriksson & Musen, 1993). Examples of such tools are DOTS (Eriksson, 1992), META•KA (Gappa, 1995), Protégé-I (Musen, 1989), Protégé-II (Eriksson, Puerta & Musen, 1994), and SIS (Kawaguchi, Motoda & Mizoguchi, 1991). Many of these metatools support the generation of knowledge-acquisition tools that acquire knowledge-base structures, such as instances and rules. In many situations, however, acquiring ontologies directly from domain experts is necessary. In Computer Science, researchers use the word *ontology* to denote several types of models (Neches, Fikes, Finin, Gruber, Senator & Swartout, 1991). In Protégé, however, *ontology* denotes models based on *class hierarchies*.

RÉSUMÉ is an example of an application of Protégé where it is insufficient to only acquire instances from domain experts (Shahar & Molina, 1998; Shahar, Chen, Stites, Basso, Kaizer, Wilson & Musen, 1998). In the RÉSUMÉ project, domain experts were only able to add and edit domain instances, not domain-specific classes, which presented a severe problem. The classic knowledge-acquisition bottleneck reappeared, because a knowledge engineer had to maintain classes by working with the expert. Thus, Protégé, as well as other metatools, have difficulties in generating knowledge-acquisition tools that integrate ontology acquisition with the acquisition of other structures, such as instances.

Until recently, the Protégé approach assumed two separate modes for ontology and instance editing. For example, the knowledge-acquisition tools generated by the Protégé tools could not manage classes. The previous versions made a clear distinction between ontology editing (which was typically the task of the developer), and editing of knowledge bases consisting of instances (which was typically the task of the domain expert). Although this distinction helped clarify the roles of the developer and domain expert, it limited the support for acquisition of classes, which are an important part of the target system. To loosen up this clear-cut distinction, we extended the Protégé approach to support the generation of ontology editors.

The new functionality for ontology acquisition allows developers to specify metalevel aspects of the input ontology (i.e., class and slot metaclasses[1]), and to generate automatically knowledge-acquisition tools that support ontology editing. The generated knowledge-acquisition tools support the editing of both classes and instances in a single tool. Moreover, knowledge-acquisition tools can contain several ontology editors that operate on different sub-ontologies (i.e., different class subtrees). Because the distinction between classes and instances is most often a modeling choice rather than an inherent property of the domain, this approach improves the modeling flexibility for the developers.

2 Background

To understand fully how the generation of ontology editors works, it is important to know the basics of the Protégé approach, and its implementation in the Protégé-II and Protégé-2000 architectures.

2.1 The Protégé Approach

The Protégé framework consists of a set of tools that support the development of knowledge-based systems. There are two major areas of support in the Protégé approach: (1) the development of problems solvers from reusable components (Eriksson, Shahar, Tu, Puerta & Musen, 1995), and (2) the generation of domain-specific knowledge-acquisition tools (Eriksson *et al.*, 1994).

The developer uses the Protégé tool set to create a domain ontology, and to select a problem-solving method that can be configured to perform the application task. The developer then uses the ontology as the basis for generating a knowledge-acquisition tool with a metatool. Domain specialists can then use this knowledge-acquisition tool to create knowledge bases, which consist of instances of the domain ontology.

The Protégé approach is to use domain ontologies as the basis for the generation of knowledge-acquisition tools. Originally, Protégé used MODEL, which is an extension of the object-oriented part of CLIPS (NASA, 1991), as the primary modeling language for ontologies

[1]In the remainder of this paper, however, we use the terms *metaclass* and *metaslot* as shorthand for class and slot metaclasses, respectively.

(Gennari, 1993; Gennari, Tu, Rothenfluh & Musen, 1994). Protégé-2000 also have support for other formats, such as OKBC (Karp, Myers & Gruber, 1995) and database storage.

2.2 The Protégé Implementations

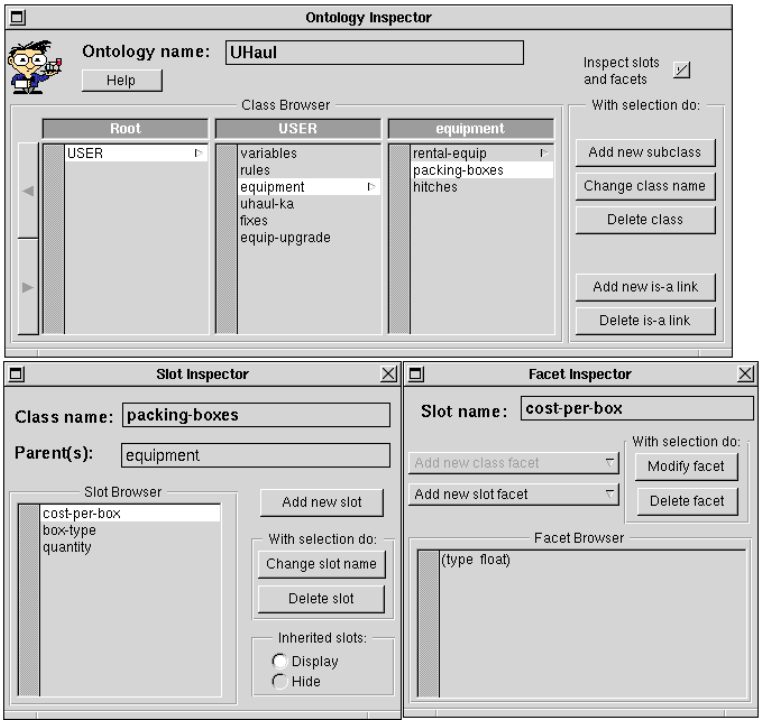
We have used two Protégé implementations, Protégé-II and Protégé-2000, as the platforms for our experiments in generating ontology editors. Protégé-II is an implementation of the Protégé approach under the NeXTstep platform, whereas Protégé-2000 is a recent Java-based implementation. In addition, there is an implementation for Microsoft Windows, Protégé/Win, which we did not use in this work. We started using Protégé-II because the other implementations were not available at the time. We then continued this work on the Protégé-2000 platform.

The Protégé tool set includes tools for ontology editing and for the generation of knowledge-acquisition tools. In Protégé-II and Protégé/Win the tools are separate applications that communicate through documents, whereas in Protégé-2000 the tools are integrated in a single application. We have modified and extended these tools to support generation of ontology editors. Let us discuss briefly the task of the Protégé tools and how the developers work with them before moving on to general ontology editing and the generation of domain-specific ontology editors.

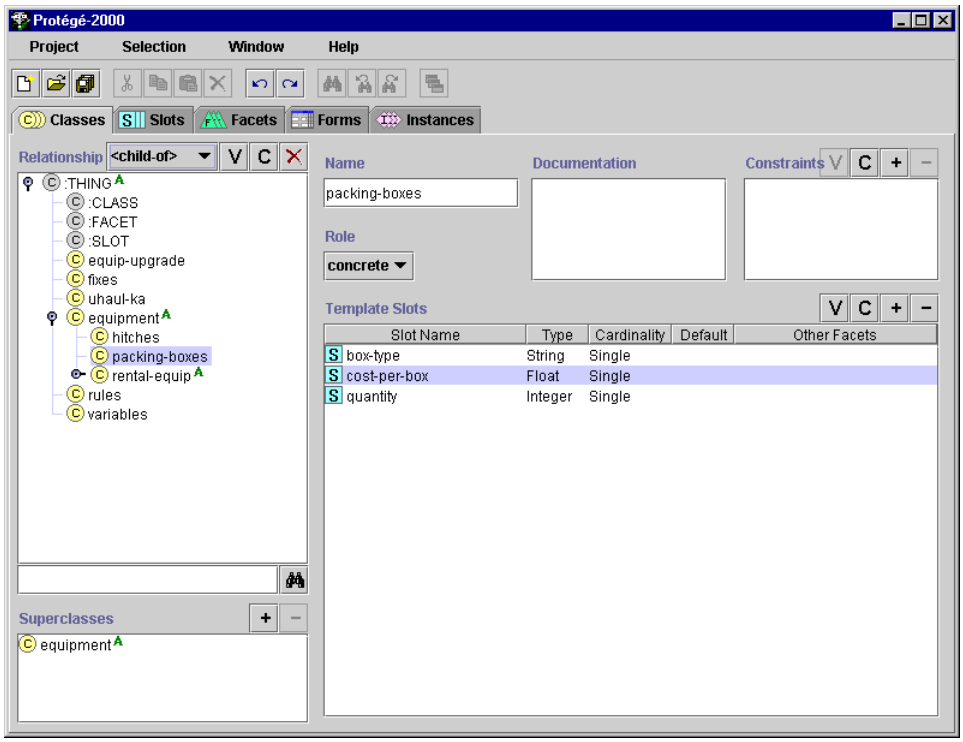
The Protégé ontology editors allow developers to edit ontologies graphically. Figure 1 shows sample screen pictures of (a) the Maître ontology editor, which is part of Protégé-II, and (b) the ontology editor in Protégé-2000.[2] In Protégé-II, the metatool DASH supports the generation of knowledge-acquisition tools from ontologies developed in Maître. Protégé-2000 is an integrated environment, and developers move among the tools by selecting different *tabs*. The ontology editor is available under the *classes* tab (see Figure 1b). Generation of knowledge-acquisition tools, including custom adjustment of forms, is available under the *forms* tab.

Note that the Protégé approach makes a clear distinction between classes and instances. Developers define the classes using the ontology editors, and the domain experts create the instances using the knowledge-acquisition tool generated. Domain specialists may find the ontology editors available difficult to use, because they are designed as general-purpose tools, and are not domain oriented. Until now, it was not possible to combine class and instance editing in the same tool. Thus, the division between class and instance editing creates difficulties in certain domains where it is important for experts to define many classes. Examples of such domains are medical domains with class hierarchies of symptoms, diseases, and drugs (Tu, Eriksson, Gennari, Shahar & Musen, 1995), technical domains with component hierarchies (Rothenfluh, Gennari, Eriksson, Puerta, Tu & Musen, 1996), and characterization-oriented domains (Domingue & Motta, 1999). Figure 2 summarizes the tool-generation process in Protégé-II. The enhancements to Protégé-II presented here allow for ontology editing in generated tools. The generation process in Protégé-2000 is similar, but the integration of the tools is much tighter. Regardless of platform, our approach to the generation of ontology editors alleviates the problem of separate ontology and instance editing.

[2] Figure 1 shows the Maître NeXTstep application and the Java-based ontology editor in Protégé-2000. In Protégé/Win the appearance of the ontology editor is somewhat different.



(a)



(b)

FIGURE 1. Protégé ontology editors. (a) The Maître ontology editor allows developers to edit the class hierarchy (upper window). (b) The Protégé-2000 environment provides a "Classes" tab that allows developers to edit the class hierarchy, and a class form that allows developers to edit class properties.

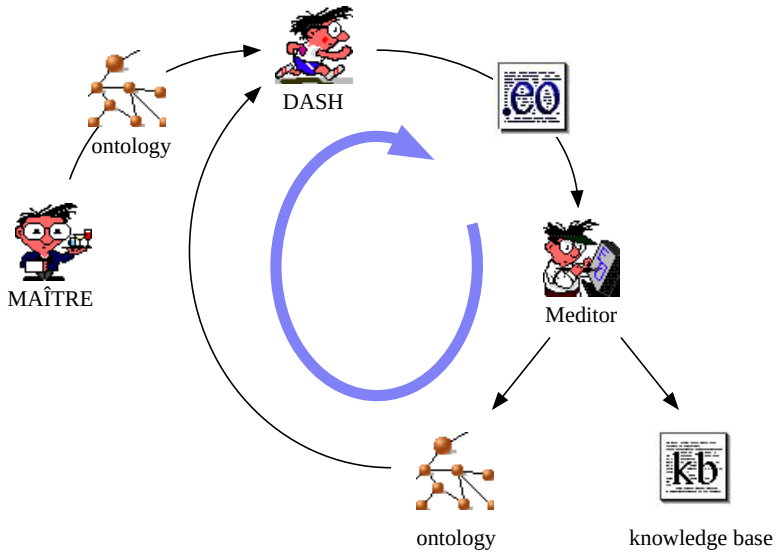


FIGURE 2. Generation of knowledge-acquisition tools and ontology editors in Protégé-II. Maître edits an ontology that is the input to the DASH metatool. DASH then produces a tool specification (EO-file), which the Meditor run-time system uses. Domain specialists can then use these tools to edit knowledge bases, and, with extensions to Protégé-II, edit ontologies. Note that ontologies defined with generated editors can be used as input to DASH.

3 Method

Developing an understanding of the issues involved in generation of ontology editors and of the appropriate tools and techniques requires experimentation with a series of prototypes. We started this research with an initial feasibility analysis. We then proceeded with prototype implementation. Rather than developing a single prototype for testing the generation of ontology editors, we have experimented with extensions to preexisting versions of Protégé.

For these experiments, we did not always use the latest research versions of Protégé (where the purpose is to explore other aspects of modeling and tool generation). Instead, we used tested and stable versions in the Protégé series (Grosso, Eriksson, Fergerson, Gennari, Tu & Musen, 1999). We used Protégé-II as the basis for the first prototype because it provided the best platform for experiments at the time. As the Protégé-2000 framework stabilized, we reimplemented the prototype in Java.

Because the introduction of support for ontology generation is a long-term effort, our approach is to refine the prototypes over time. We have analyzed the advantages and disadvantages of each generation and have incorporated improvements to the subsequent generation. Figure 3 shows the series of prototypes for ontology generation we have developed.

4 Ontology Editors

Several ontology editors are available. Let us examine a selection of related approaches to ontology editing. Many development environments for object-oriented languages, such as SmallTalk 80 (Goldberg & Robson, 1983) and C++, include tools for ontology editing. In these environments, developers can edit class hierarchies graphically. Such editors, however, are designed to be used by programmers and are not primarily intended for non-programmers

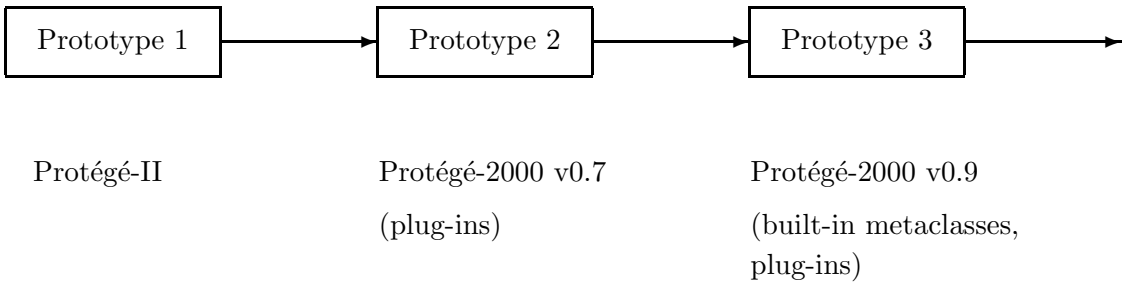


FIGURE 3. Series of prototypes. This project started using Protégé-II as a basis and then continued using recent versions of Protégé-2000.

(e.g., domain experts). In addition to these language-oriented tools, there are ontology editors based on knowledge-engineering and knowledge-acquisition principles.

The Ontolingua server (Farquhar, Fikes & Rice, 1997) is a set of tools that support ontolingua ontology development on the world-wide web. These tools allow developers to browse, edit and publish ontologies using web technologies (e.g., CGI scripts and Java).

The Ontosaurus browser (Swartout, Patil, Knight & Russ, 1996) is a similar web-based ontology editor. One of the significant differences between the Ontosaurus and Ontolingua servers is that the former aims at supporting distributed collaborative development. Another example of an ontology editor for collaborative ontology development is the HITS Knowledge Editor (Terveen & Wroblewski, 1990).

Open Knowledge Base Connectivity (OKBC) is a programming interface that supports the accessing of frame-based knowledge bases (Karp *et al.*, 1995). The Generic Knowledge Base Editor (GKB-Editor) is an ontology editor that uses OKBC to manipulate the ontology representation.

CODE4 (Skuce & Lethbridge, 1995) is an ontology editor and knowledge management system that allows developers to create ontologies with inheritance, property, statement, relation, and facet hierarchies. CODE-4 has an advanced user interface that provides several views on the knowledge structures edited.

KARO (Pirlein & Studer, 1995) is an approach that provides semantic means for reusing ontologies. This approach emphasizes retrieval and adaptation of reusable ontologies, and the construction of new ontologies from such components.

Rational Rose (Quantrani, 1998) is a commercial visual-modeling environment that allows developers to model conceptual structures and to generate source code (stubs) in different programming languages from them. Rose provides four major views: the *use-case, logical, component*, and *deployment* views. The logical view is equivalent to an ontology editor that supports the drawing of class diagrams.

In summary, the goal of the ontolingua-server, and OKBC approaches is to enable knowledge sharing through reusable ontologies. The web-based ontolingua editor is an example of how tools can support collaborative ontology development. CODE4, KARO, and Rational Rose are examples of tools that provide advanced domain-independent support for ontology editing. Thus, an important difference between these ontology editors and the Protégé approach is that an essential requirement for Protégé is to generate knowledge-acquisition tools rather that support ontology editing. The design of the Protégé tools is influenced by this aim of supporting the knowledge acquisition from domain experts, whereas the other tools, such as OKBC and Rational Rose, are designed primarily for developers and programmers.

```
(defclass automobile (is-a USER)
       (slot registration-number (type string)
            (ka-specification browser-key))
       (slot make (type string))
       (slot year (type integer))
       (slot owner (type instance)
            (allowed-classes person))
   )
```

FIGURE 4. Sample class definition. The definition of the class `automobile` includes slot definitions for registration number, make, year, and owner. Although this figure shows the textual class definition, developers can use the Protégé ontology editors to define the class graphically.

Another major difference between the tools discussed in this section and the Protégé approach is that Protégé supports domain-specific editors. Although some editors allow their users to change the presentation of the ontology, they do not provide truly domain-oriented views. Protégé, however, does not currently provide advanced editing support in the sense that it does not support consistency checking and natural-language understanding, for example.

5 Specification of Ontology Editors

The ontology editing facility in Protégé-II allows the developer to define metaclasses and metaslots in the input ontology to DASH. (Although the ontology-definition language used in Protégé-II does not support metaclasses and metaslots per se, we can *annotate* the input ontology to DASH with specifications for ontology editing, metaclasses, and metaslots.) The ontology editor in Protégé-2000 supports built-in metalevel classes, slots, and facets, which are the basis for the generation of ontology editors.

5.1 ONTOLOGIES IN PROTÉGÉ-II

Before we discuss metaclasses and metaslots, let us examine how developers define ontologies (i.e., classes and slots) in Protégé and MODEL.

5.1.1 Classes, Slots, and Facets

Figure 4 shows a sample class definition for automobiles. The class `automobile` is a subclass of `USER,` which is a predefined *root class* in CLIPS . Furthermore, `automobile` consists of the slot definitions `registration-number make, year,` and `owner`. The slot definitions consist of the slot name and a list of slot *facets,* which are the properties of the slot. Examples of such facets are slot type and default value. In Figure 4, the slot `make` is of type string; that is, the slot facet `type` has the value `string`. The slot `owner` is a pointer to an instance (`(type instance)`) and the allowed class for this instance is `person` (`(allowed-class person)`)[3].

[3]The class `person` is not shown.

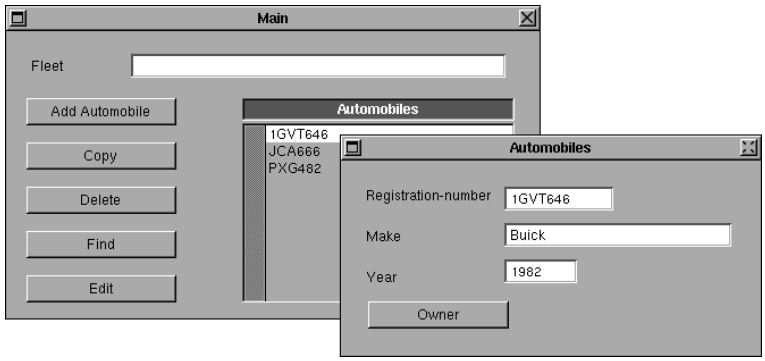


FIGURE 5. The resulting forms generated by DASH. The form on the right corresponds to the class `automobile`. The form on the left contains a list browser with the automobiles entered. This browser shows the values of the `registration-number` slot for each automobile (because this slot is the browser key).

5.1.2 Ontology Annotations: Facets for Knowledge Acquisition

Protégé-II supports *ontology annotations* through additional slot facets for the specification of knowledge-acquisition information. Because the generation of knowledge-acquisition tools from ontologies sometimes requires additional information about slots (i.e., slot annotations), developers can use the `ka-specification` slot facet to provide this information. DASH uses this information to generate the appropriate user-interface widgets in the target tool. For example, in Figure 4, the slot `registration-number` has the `ka-specification` facet set to `browser-key`. DASH will then assume that the tool users will browse the automobiles by their registration number, and display the registration numbers in list browsers, and so on. Figure 5 exemplifies the use of the browser key in the tool generated.

An alternative approach to ontology annotations is to provide this information at a later tool-generation stage, such as during layout custom tailoring. Protégé-2000 uses this approach. In general, annotation-free ontologies are cleaner than annotated ones, because they do not mix domain and tool models. In practice, however, ontology annotations work well, because they are straightforward to maintain (i.e., the annotations always follow changes to the ontology code and do not require a special maintenance mechanism). They are less flexible than specification at generation time, because you cannot generate substantially different knowledge-acquisition tools from the ontology (since the annotations used to guide tool generation are in the ontology).

5.2 METALEVEL OBJECTS

Let us briefly consider the concept of metalevel objects. In object-oriented programming, *metaclasses* are specification classes (i.e., they model class properties rather than object properties). A metaclass definition typically includes *slots* that model the class name, superclasses, subclasses, list of slots, and additional class features. In turn, *metaslots* are specifications of the slots used and their properties (facets). Metaslot definitions often include slot name, slot type, default value, and so on.

Advanced object-oriented languages, such as SmallTalk 80 (Goldberg & Robson, 1983) and CLOS (Keene, 1989), allow programmers to extend the language by subclassing, and thus modifying, intrinsic metaclasses (Kiczales, des Rivières & Bobrow, 1991). In such languages, it is possible to change the behavior of classes, including the inheritance model for subclassing.

5.3 Metalevel Objects in Model

The version of MODEL used in Protégé-II does not support metalevel objects. This restriction complicates the generation of ontology editors, because we must emulate metaclasses, metaslots, and so on using standard MODEL objects. In Protégé-II, we use ontology annotations to specify that DASH should treat certain classes as metaclasses (see Section 5.5.1).

The Protégé-2000 approach is to add intrinsic metalevel objects to MODEL. Because the MODEL implementation in Protégé-2000 is based on Java rather than on extensions to the CLIPS implementation (as in Protégé-II), it is possible to support true metalevel classes, slots, and facets. The class representation and the structure of the metaclass hierarchy was inspired directly by the CLOS approach (Kiczales *et al.*, 1991). However, adding intrinsic metalevel objects to Protégé-II would require a fundamental reengineering of the CLIPS-based MODEL implementation.

5.4 Tree Editors

In addition to specifying ontology editors using metalevel objects, we must provide user-interface widgets that support ontology editing in the target tools. Thus, we require the metatool to support at least a minimal set of user-interface components for ontology editing.

The domain-specific ontology editors in Protégé-II are based on the concept of *tree editors*. In Protégé-II, tree editors are analogous to list browsers and graph editors, in the sense that they operate on sets of objects (Eriksson, Puerta, Gennari, Rothenfluh, Tu & Musen, 1995). In this approach, the tree editors are *multicolumn list browsers* where the most general class is displayed in the left-most column and where subclasses are displayed in the subsequent columns. (See Figure 1a for an example of a multicolumn browser that operates on a tree, in this case a class hierarchy in Maître.)

Protégé-2000 uses a tree editor based on Java's `JTree` class. We have adapted the appearance and functionality of this tree editor to ontology editing (see Figure 1b). Because Protégé-2000 supports plug-in components, it is possible to replace the standard tree implementation with a custom-tailored one. Such changes, however, require Java programming.

Note that the semantics of basic tree editors is insufficient for ontology editing. For instance, slot inheritance is not supported in tree editors per se. Another difference between the tree and ontology editors is that the output of tree editors consists of instances in the knowledge base, whereas the output of ontology editors consists of class definitions in the output ontology.

5.5 Ontology Specification

To specify ontology editors, we must take advantage of the concepts of metaclasses and metaslots (see Section 5.2). It is possible to think of metaclasses as specifications for the *class editors* in the ontology editor. In the Protégé approach, the definition of a metaclass specifies the corresponding class editor. A minimal metaclass definition often includes the class name, pointers to superclasses or subclasses, and a list of slots. The corresponding class editor can then consist of a text field for entering the class name, and a list browser for the list of slots.

5.5.1 Protégé-II Specification

As discussed in Section 5.1.2, Protégé-II uses ontology annotations to specify the metalevel objects. The slot facet `(ka-specification ontology)` indicates that the slot points to the

```
(defclass class (is-a USER)
       (slot class-name (ka-specification browser-key)
            (type string))
       (slot slots (allowed-classes model-slot)
            (type instance)
            (cardinality multiple))
       (slot sub (allowed-classes class)
            (type instance)
            (cardinality multiple)
            (ka-specification tree-link))
   )


(defclass model-slot (is-a USER)
       (slot slot-name (ka-specification browser-key)
            (type string))
       (slot type (allowed-symbols symbol string integer float boolean instance)
            (type symbol)
            (default string))
   )
```

FIGURE 6. Sample metaclass and metaslot definitions in Protégé-II. In the metaclass `class`, the slot `slots` holds a list of metaslot definitions (i.e., instances of the class `model-slot`). The slot `sub` contains a list of the subclasses. In this case, the allowed subclass instances must be of the same metaclass (i.e., `class`). In the metaslot `model-slot`, the slots `slot-name` and `type` hold the slot name and type, respectively.

top-level classes in the ontology editor. When Protégé-II recognizes this slot facet, it assumes that the tree should be edited as an ontology using the appropriate semantics. Protégé-II assumes that these top-level classes (i.e., the allowed classes for this slot) are the metaclasses for the ontology editor.

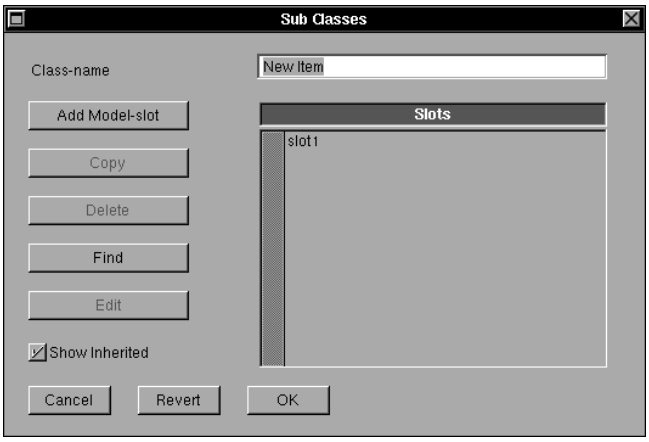
Figure 6 includes a sample metaclass definition that can be used as the basis for a class editor. The class named `class` defines slots for the class name, the slot of the class, and the subclasses. Figure 7a shows the resulting class editor. As specified in the metaclass definition, the class editor contains a text field for the class name and a list browser for slots. The user can use the browser and its control buttons to manage the slot definitions. (In this example, the class definition contains a single local slot, "slot1".)

In Protégé-II, a metaslot definition is a class that defines the slot facets. Figure 6 includes an example of a metaslot definition that includes the slot name and a *type* facet. The allowed symbols for the slot `type` specify the possible slot types. Figure 7b shows the resulting slot editor. The user can use this editor to specify the name for the slot using a text field, and the data type of the slot using a pop-up menu.

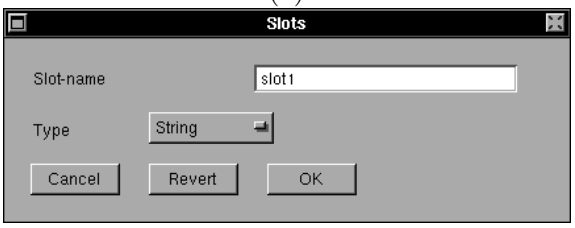
5.5.2 Protégé-2000 Specification

In Protégé-2000, developers define metaclasses by subclassing metaclasses. Following OKBC, Protégé-2000 provides the built-in metaclass `:CLASS`. All subclasses of `:CLASS` are metaclasses. Figure 8 shows the subclassing of the built-in metaclass `:CLASS`. The subclass `my-meta-class` is the user-defined metaclass.

For the specification of metalevel slots and facets, Protégé-2000 provides the built-in classes



(a)



(b)

FIGURE 7. Sample class and slot editors. (a) The class editor contains a the list browser that shows the slots defined in this class. The buttons on the left-hand side allow the user to manage the list of slots. (b) The slot-editor form allows users to edit the slot name and type.

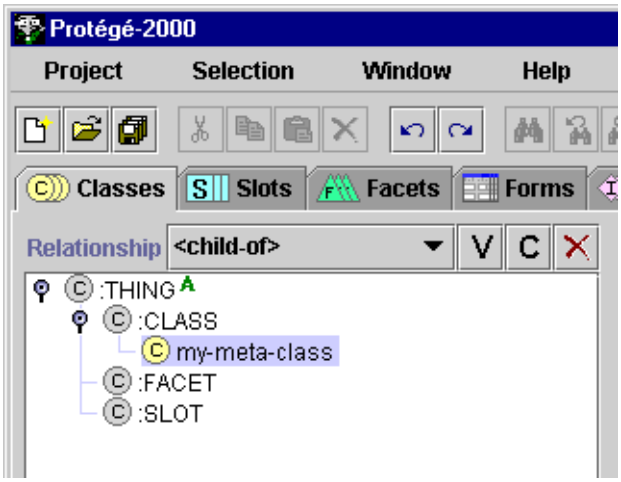


FIGURE 8. Sample metaclass definition in Protégé-2000. Developers create new metaclasses by subclassing `:CLASS`.

`:SLOT` and `:FACET` to subclass from. Although this functionality is not currently implemented, the model allows for subclasses of `:SLOT` and `:FACET` to be used as templates for slots and facets in new metaclasses. In addition, Protégé-2000 has tabs for slots and facets. Slots attached to metaclasses (e.g. `my-meta-class`) also appear under the slots tab. In Protégé-2000, there is no difference between regular slots and metalevel slots. The metaslots are slots attached to metaclasses.

Unlike Protégé-II, Protégé-2000 does not support the notion of ontologies in the sense that developers define an explicit ontology class. The Protégé-2000 architecture integrates ontologies and knowledge bases (i.e., instances) in the sense that the classes and instances coexist in the system. For example, classes appear under the instances tab, because they are instances of metaclasses. Likewise, metaclasses appear under the instances tab, because they are instances and subclasses of `:CLASS`. Because the Protégé-2000 architecture allows developers to create plug-in modules for new tabs, it is possible to control the users' view by implementing new tabs with the ontology and knowledge-base views required.

5.5.3 Specification Conclusion

In summary, the Protégé approach provides a flexible mechanism for the specification of ontology, class, and slot editors. By adding new slots to the metaclass and metaslot definitions, developers can extend the set of class and slot facets supported. Because one of the cornerstones of the Protégé approach is to allow the developers to custom tailor form layout (Eriksson *et al.*, 1994), developers can use this functionality to adjust the ontology, class, and slot editors.

6 Examples

Let us consider examples of ontology editors generated by Protégé-II and Protégé-2000.

6.1 Ontology editors generated by PROTÉGÉ-II

We begin by examining an ontology editor generated by Protégé-II. Figure 9 shows a generated ontology editor that is based on Maître. The ontology used as the basis for this editor consists of metaclass and metaslot definitions that correspond to the MODEL language. The metaclass definition is the basis for the class editor (see Figure 9, lower left-hand side). The metaslot definition is the basis for the slot editor (see Figure 9, lower right-hand side).

This example shows that it is possible to use Protégé-II to generate domain-independent editors similar to Maître. By starting with general metaclass and metaslot definitions, and by modifying them gradually, developers can change general ontology editors into domain-specific ones as required by the knowledge-acquisition process.

6.2 Ontology editor generated by Protégé-2000

Developers use the classes tab in Protégé-2000 to define metaclasses. Figure 10 shows the subclassing of the built-in metaclass `:CLASS` in Protégé-2000. In this example, we simply define `my-meta-class` as a subclass of `:CLASS` without any new slots. The purpose is to generate a class editor with a custom-tailored layout.

The forms tab in Protégé-2000 supports editing of class forms as well as forms for instances. Figure 11 shows the form editor for `my-meta-class`. We use this form editor to custom tailor the layout for the class editor. Compared to the standard class-editor layout, the layout

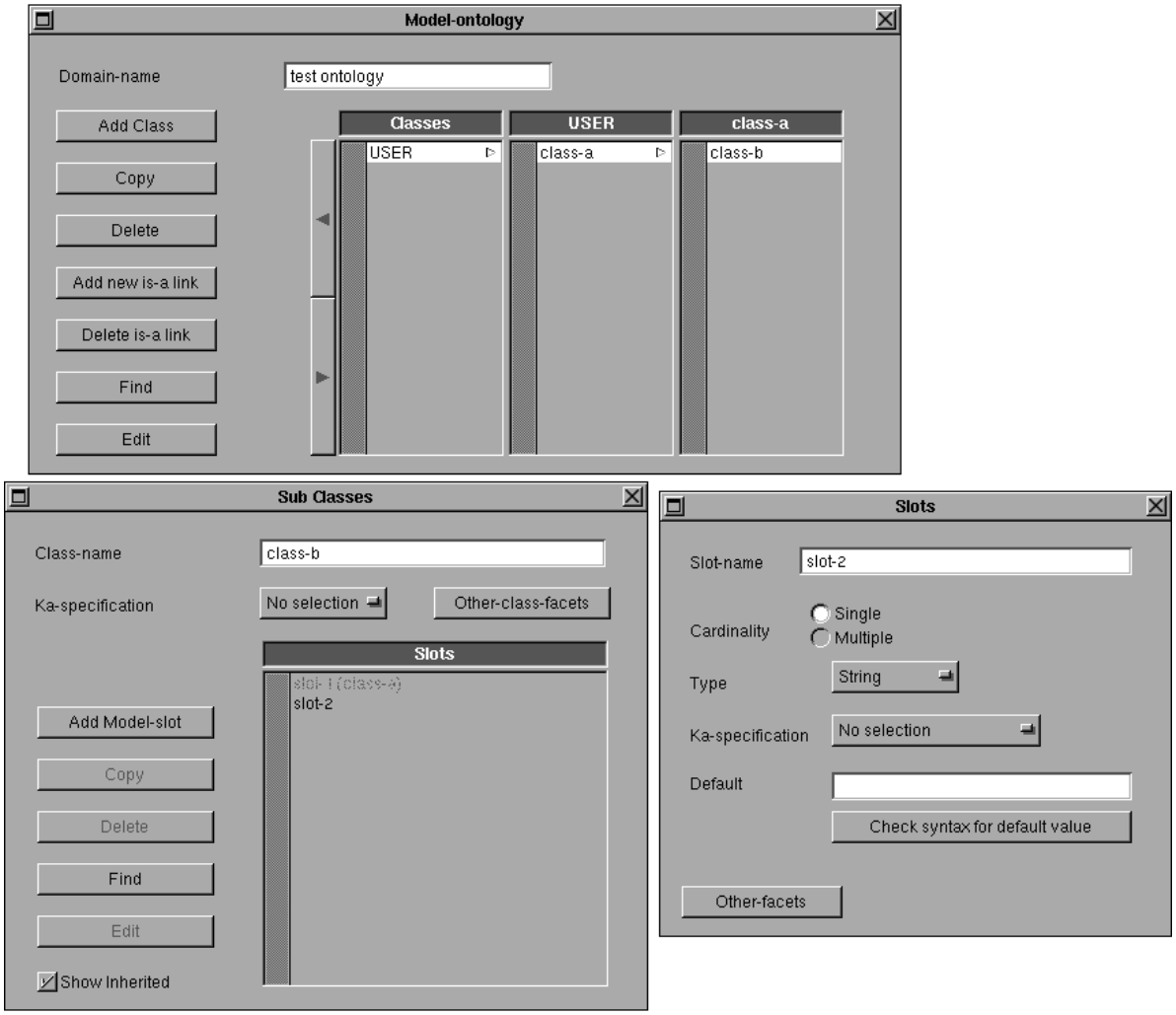


FIGURE 9. Generated ontology editor based on the design of Maître. The basis for this editor consists of metaclass and metaslot definitions that represent the MODEL implementation of classes and slots. Therefore, the ontology editor is domain independent.

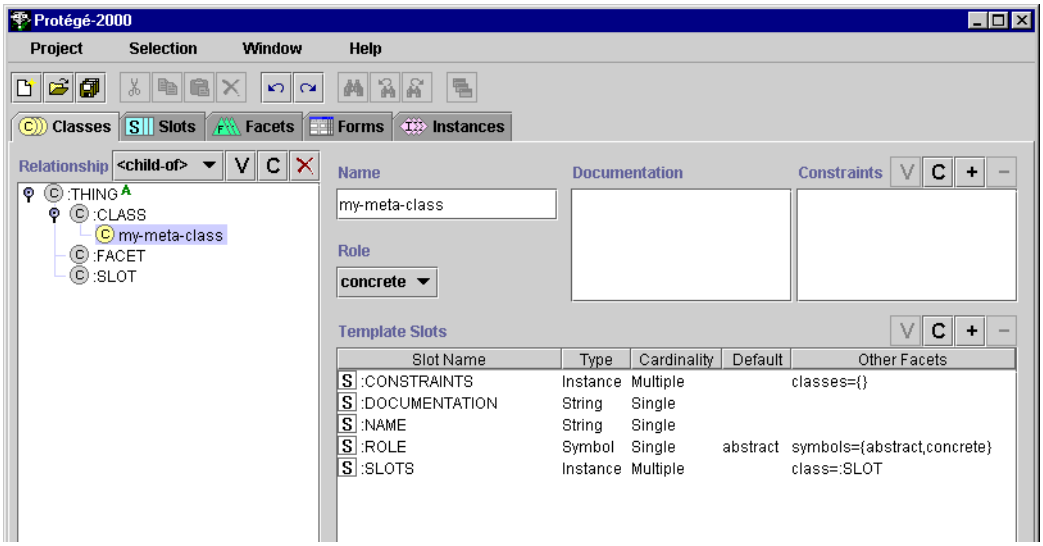


FIGURE 10. Definition of metaclasses. This configuration includes the built-in classes `:CLASS`, `:FACET`, and `:SLOT`. It is possible to create new metaclasses by subclassing `:CLASS`. The sample metaclass `my-meta-class` is a subclass of `:CLASS`.

in Figure 11 has the *role* field to the right of the *name* field and the *documentation* and *constraints* fields at the bottom.

Let us consider the situation where we add a new property to the metaclass. Figure 12 shows `my-meta-class` with the added metaslot `author`. Classes with the *author* property can then store textual information about the name of the class author. Moreover, the added `author` metaslot results in an author field on the class form. Figure 13 shows the form editor with the author field. This form will then support editing of class definitions with author properties.

7 Discussion

As stated in Section 1, the strict separation between ontology definition and knowledge-base editing in the standard versions of Protégé-II, Protégé/Win, and Protégé-2000 poses significant restrictions on the knowledge-acquisition process. Although we initially thought of ontology editing as a task for developers rather than for domain experts, the use of generated tools has highlighted the need for ontology editing by domain experts in many situations. The generation of ontology editors redefines the role of domain experts in the Protégé approach in the sense that we can allow experts to take responsibility for parts of the ontology-definition task. Let us discuss how we can take advantage of the new combination of ontology and instance editing.

7.1 Combining Ontology and Instance Editing

The combination of ontology and instance editing in integrated knowledge-acquisition tools provides many possibilities in terms of tool design. Figure 14 illustrates three approaches to ontology editing and knowledge acquisition. The basic configuration of an ontology editor with a class editor (Figure 14a) can be extended with instance editing in additional fields

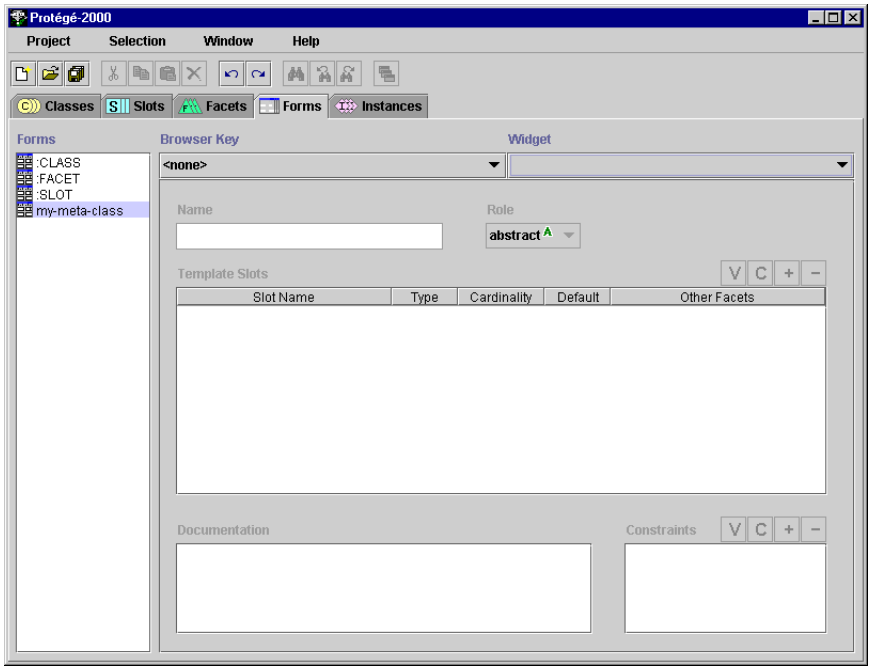


FIGURE 11. Custom adjustment of forms corresponding to metaclasses. The developer can use the forms editor to adjust the layout for custom metaclasses. This sample form shows the layout for `my-meta-class`. Note that this layout is different from the standard class-definition form (see Figure 10).

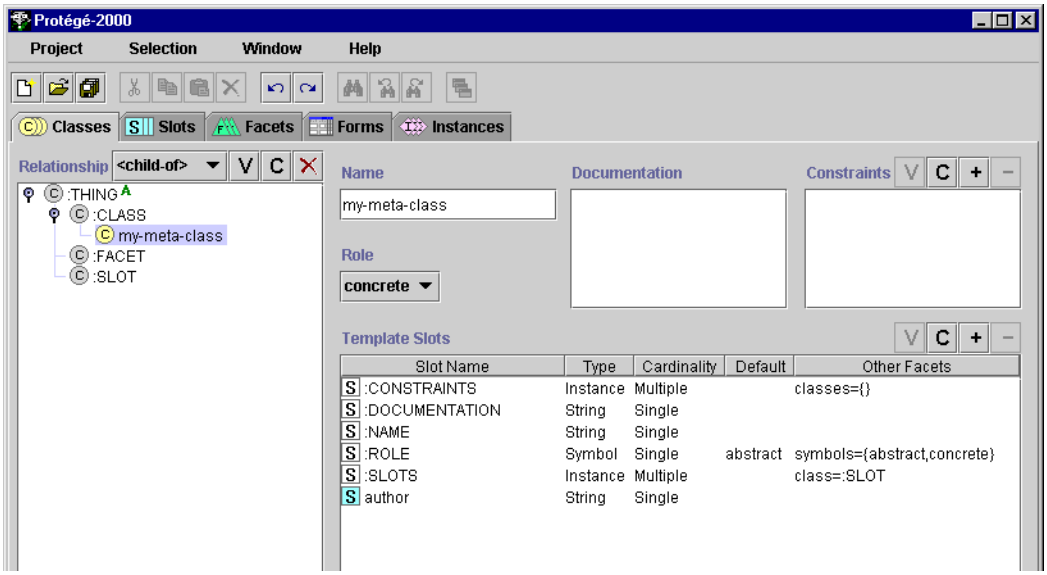


FIGURE 12. Addition of the metaslot `author` to `my-meta-class`.

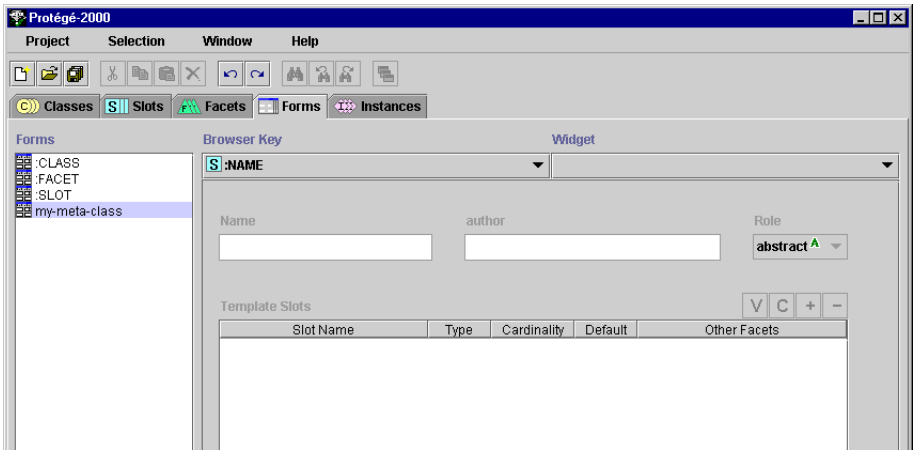


FIGURE 13. Custom adjustment of forms with the author field. This sample form shows the layout for `my-meta-class` with the `author` metaslot.

(Figure 14b). Also, it is possible to use several ontology editors in the same knowledge-acquisition tool (Figure 14c). The latter approach is especially useful when there are several relevant ontologies in the domain (e.g., separate symptom, disease, and drug ontologies in a medical domain). In summary, the combination of ontology editors and knowledge-acquisition tools enables the generation of well-integrated development tools.

The intial Protégé assumption was that ontology development is the task of the knowledge engineer and that population of the knowledge base (with instances) is the task of the domain expert. Protégé applications, such as RÉSUMÉ, have shown that this assumption constrains the use of the knowledge-acquisition tools generated. However, there is a trade-off between a dynamic approach, where advanced users can change everything, and easy of use for new users (e.g., by controlled functionality).

The combined ontology and instance editing presented here would certainly alleviate the difficulties that arise in projects such as RÉSUMÉ. RÉSUMÉ, however, requires other extensions to Protégé, such as multiple-dimension tables, inverted links, and type checking (Shahar & Molina, 1998; Shahar *et al.*, 1998). It is possible to implement some of this functionality through plug-in modules in Protégé-2000.

Although the support for domain-oriented ontology editing in Protégé-II and Protégé-2000 is a significant improvement in the usability of the target tools, there are certain shortcomings in the current implementations. For instance, it is possible to improve general ontology editing in Protégé-II and Protégé-2000 by adding support for the editing of class hierarchies as *graphs*.

7.2 GRAPH EDITING

Although the multicolumn browsers used for class-hierarchy editing are sufficient in most cases, there are situations when graph-based editors are more appropriate than multicolumn browsers. Kremer (1996; 1997) has developed graph editors for concept maps where it is possible to define ontology graphs that specify the possible instance graphs. This approach has the advantage that the developer can define metalevel (ontology) graphs and actual (instance) graphs on the same canvas. Unlike Protégé, however, this system does not provide instance editing in forms, and so on. In principle, it is possible to extend the Protégé implementations to support graph-based editing of class hierarchies (e.g., by taking advantage of the graph editors supported in the knowledge-acquisition tools).

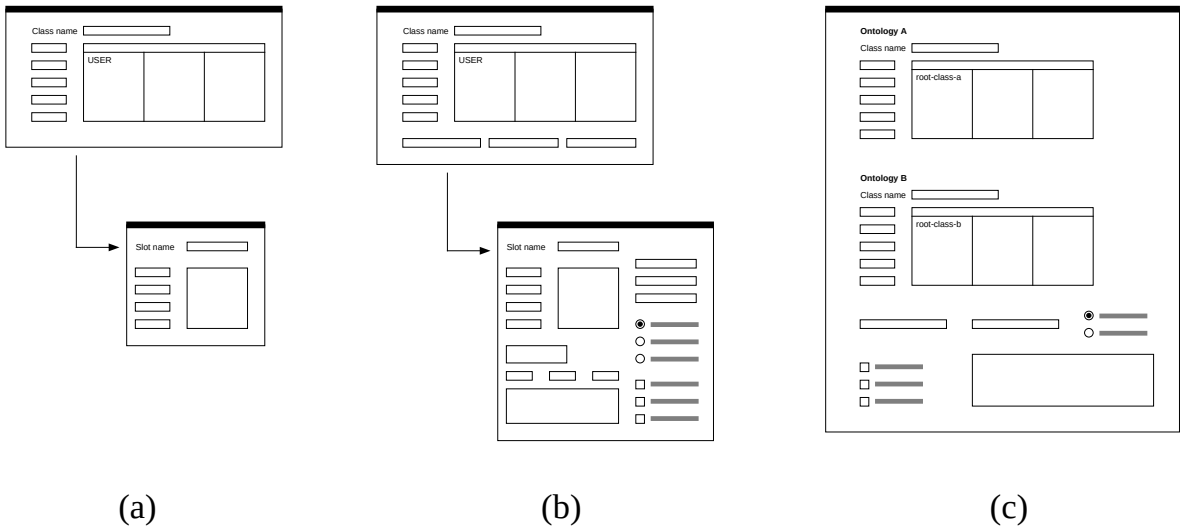


FIGURE 14. Different combinations of ontology editors and knowledge-acquisition tools. (a) Stand-alone ontology editor (e.g., a domain-independent ontology editor). (b) Ontology editor extended with knowledge-acquisition components (for editing of instances). (c) Knowledge-acquisition tool with two integrated ontology editors (for Ontologies A and B). These ontology editors support the definition of two separate ontologies from the root classes `root-class-a` and `root-class-b` respectively.

7.3 METALEVEL OBJECTS

Because the original version of MODEL does not support intrinsic metalevel objects, we must add this functionality. In the Protégé-II version, we used ontology annotation and sophisticated widgets as the basis for the generation of ontology editors (see Section 5.5.1). In Protégé-2000, we extended MODEL with metalevel objects, which make the introduction of ontology-editor generation much cleaner.

Although MODEL is an appropriate ontology language for Protégé at present, we recognize that developers will request support for other ontology languages in the future. Such ontology languages may not include support for metalevel objects. Thus, metalevel objects to MODEL will not automatically remove the need for metalevel annotation in every situation.

Protégé-II uses special versions of the user-interface widgets for ontology editing. These widgets implement the semantics for inheritance. For example, the list-browser widget displays the slot inherited from superclasses as well as the local slots. In this case, the widget interacts with other widgets to compute the list of inherited slots. In the Protégé-2000 approach, however, we use the inheritance model provided in the metaclass definitions as the basis for the editor rather than to hard code the semantics in the widgets.

The fact that Protégé-2000 makes it possible to edit the class and slot forms is a natural result from intrinsic metalevel objects (where classes and slots are instances of their respective metaclass). Because Protégé-2000 allows custom tailoring of forms for the acquisition of instances, no additional work or special casing is required when *classes are instances*. Likewise, the ability to alter the metaclass (and thus custom tailor the creation of classes) is a direct result of making the metaclass a *class like any other class* in the system.

7.4 Bootstrapping Ontology Editors

An interesting feature of the Protégé-II and Protégé-2000 approaches is that it is possible to *bootstrap* ontology editors by first generating a basic ontology editor and then using it to maintain and extend itself (see Figure 2). By supporting the generation of high-quality ontology editors, it is possible to replace hand-coded ontology editors by generated ones. This approach can reduce development effort and simplify the architecture of the development environment.

The metatool DOTS illustrates that it is possible to bootstrap complete metatools (Eriksson, 1992). DOTS allows the developer to specify graph editors, which can edit class hierarchies.

8 Conclusions

The enhancements to Protégé-II and Protégé-2000 that allow developers to create knowledge-acquisition tools with ontology editors illustrate how metatools can support the generation of custom-tailored ontology editors. Knowledge-acquisition tools based on domain ontologies can integrate ontology editors based on metaclass and metaslot specifications. The seamless integration of domain-oriented ontology editors in knowledge-acquisition tools allows experts to enter domain knowledge without having to use a generic ontology editor. Furthermore, the experts do not have to be concerned with the distinction between class and instance definitions.

The Protégé-2000 architecture is based on a component architecture, which makes it possible to specialize ontology editors further by creating custom-tailored widgets for ontology editors. This approach makes Protégé-2000 a suitable platform for further experiments with advanced ontology editors, and the lessons learned from such research can then be incorporated in new Protégé architectures.

Acknowledgments

This work has been supported by the Swedish Research Council for Engineering Sciences (TFR) grant no. 95-186.

We thank Magnus Bång, John H. Gennari, Jan Olsson, Klas Orsvärn, Angel R. Puerta, Thomas E. Rothenfluh, Adam Stein, and Samson W. Tu for valuable discussions and suggestions on ontology editing in Protégé-II. Ivan Rankin provided editorial assistance. On-line information about Protégé-II is available at `http://smi.stanford.edu/projects/protege/`.

References

Domingue, J. & Motta, E. (1999). A knowledge-based news server supporting ontology-driven story enrichment and knowledge. In Fensel, D. & Studer, R., Eds., *Proceedings of the 11th European Workshop on Knowledge Acquisition, Modelling, and Management*, pp. 103–120, Dagstuhl Castle, Germany. Also available as `http://kmi.open.ac.uk/~john/papers/planet_onto.pdf`.

Eriksson, H. (1992). Metatool support for custom-tailored domain-oriented knowledge acquisition. *Knowledge Acquisition*, **4**, 445–476.

Eriksson, H. & Musen, M. A. (1993). Metatools for knowledge acquisition. *IEEE Software*, **10**, 23–29.

Eriksson, H., Puerta, A. R., Gennari, J. H., Rothenfluh, T. E., Tu, S. W. & Musen, M. A. (1995). Custom-tailored development tools for knowledge-based systems. In *Proceedings of the Ninth Banff Knowledge Acquisition for Knowledge-Based Systems Workshop*, pp. 26.1–26.19, Banff, Canada.

Eriksson, H., Puerta, A. R. & Musen, M. A. (1994). Generation of knowledge-acquisition tools from domain ontologies. *International Journal of Human–Computer Studies*, **41**, 425–453.

Eriksson, H., Shahar, Y., Tu, S. W., Puerta, A. R. & Musen, M. A. (1995). Task modeling with reusable problem-solving methods. *Artificial Intelligence*, **79**, 293–326.

Farquhar, A., Fikes, R. & Rice, J. (1997). The ontolingua server: A tool for collaborative ontology construction. *International Journal of Human–Computer Studies*, **46**, 707–728.

Gappa, U. (1995). *Grafische Wissensakquisitionssysteme und ihre Generierung* [Graphical Knowledge-Acquisition Systems and their Generation]. PhD thesis DISKI 100, University of Karlsruhe, Karlsruhe, Germany. ISBN 3-89601-100-6.

Gennari, J. H. (1993). A brief guide to MAÎTRE and MODEL: An ontology editor and a frame-based knowledge representation language. Technical Report KSL-93-46, Knowledge Systems Laboratory, Stanford University, Stanford, CA.

Gennari, J. H., Tu, S. W., Rothenfluh, T. E. & Musen, M. A. (1994). Mapping domains to methods in support of reuse. *International Journal of Human–Computer Studies*, **41**, 399–424.

Goldberg, A. & Robson, D. (1983). *Smalltalk-80: The Language and its Implementation*. Reading, Massachusetts: Addison-Wesley.

Grosso, W. E., Eriksson, H., Fergerson, R. W., Gennari, J. H., Tu, S. W. & Musen, M. A. (1999). Knowledge modeling at the milliennium (The design and evolution of Protégé-2000). In *Proceedings of the Twelfth Workshop on Knowledge Acquisition, Modeling, and Management*, Banff, Canada.

Karp, P. D., Myers, K. L. & Gruber, T. (1995). The generic frame protocol. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI'95*, pp. 768–774, Montréal, Québec, Canada.

Kawaguchi, A., Motoda, H. & Mizoguchi, R. (1991). Interview-based knowledge acquisition using dynamic analysis. *IEEE Expert*, **6**, 47–60.

Keene, S. E. (1989). *Object-Oriented Programming in COMMON LISP: A Programmer's Guide to CLOS*. Massachusetts: Addison-Wesley.

Kiczales, G., des Rivières, J. & Bobrow, D. G. (1991). *The Art of the Metaobject Protocol*. Cambridge, MA: The MIT Press.

Kremer, R. (1996). Toward a multi-user, programmable web concept mapping "shell" to handle multiple formalisms. In *Proceedings of the Tenth Banff Knowledge Acquisition for Knowledge-Based Systems Workshop*, pp. 48.1–48.20, Banff, Canada. Also available as `http://ksi.cpsc.ucalgary.ca/KAW/KAW96/kremer/kremer.html`.

Kremer, R. (1997). *Constraint Graphs: A Concept Map Meta-Language*. PhD thesis, Department of Computer Science, University of Calgary, Calgary, Alberta, Canada. Also available as `http://www.cpsc.ucalgary.ca/~kremer/dissertation/`.

Musen, M. A. (1989). An editor for the conceptual models of interactive knowledge-acquisition tools. *International Journal of Man–Machine Studies*, **31**, 673–698.

NASA. Software Technology Branch, Lyndon B. Johnson Space Center, NASA (1991). *CLIPS Reference Manual*, Houston, TX.

Neches, R., Fikes, R., Finin, T., Gruber, T., Senator, T. & Swartout, W. (1991). Enabling technology for knowledge sharing. *AI Magazine*, **12**, 36–56.

Pirlein, T. & Studer, R. (1995). An environment for reusing ontologies within a knowledge engineering approach. *International Journal of Human–Computer Studies*, **43**, 945–965.

Quantrani, T. (1998). *Visual Modeling with Rational Rose and UML*. Reading, MA: Addison-Wesley.

Rothenfluh, T. E., Gennari, J. H., Eriksson, H., Puerta, A. R., Tu, S. W. & Musen, M. A. (1996). Reusable ontologies, knowledge-acquisition tools, and performance systems: PROTÉGÉ-II solutions to Sisyphus-2. *International Journal of Human–Computer Studies*, **44**, 303–332.

Shahar, Y., Chen, H., Stites, D. P., Basso, L., Kaizer, H., Wilson, D. M. & Musen, M. A. (1998). Semiautomated acquisition of clinical temporal-abstraction knowledge. Technical Report SMI-98-0735, Stanford Medical Informatics. Also available as `http://www-smi.stanford.edu/pubs/SMI_Reports/SMI-98-0735.pdf`.

Shahar, Y. & Molina, M. (1998). Knowledge-based spatiotemporal linear abstraction. *Pattern Analysis and Applications*, **1**, 91–104.

Skuce, D. & Lethbridge, T. C. (1995). CODE4: A unified system for managing conceptual knowledge. *International Journal of Human–Computer Studies*, **42**, 413–451.

Swartout, B., Patil, R., Knight, K. & Russ, T. (1996). Toward distributed use of large-scale ontologies. In *Proceedings of the Tenth Banff Knowledge Acquisition for Knowledge-Based Systems Workshop*, pp. 32.1–32.19, Banff, Canada. Also available as `http://ksi.cpsc.ucalgary.ca/KAW/KAW96/swartout/Banff_96_final_2.html`.

Terveen, L. G. & Wroblewski, D. A. (1990). A collaborative interface for editong large knowledge bases. In *Proceedings of the Eighth National Conference on Artificial Intelligence, AAAI-90*, pp. 491–496.

Tu, S. W., Eriksson, H., Gennari, J. H., Shahar, Y. & Musen, M. A. (1995). Ontology-based configuration of problem-solving methods and generation of knowledge-acquisition tools: Application of PROTÉGÉ-II to protocol-based decision support. *Artificial Intelligence in Medicine*, **7**, 201–225.